PET

PET

PET

PET

PERSONAL COMPUTER

PERSONAL COMPUTER

PERSONAL COMPUTER

PERSONAL COMPUTER

PET USER MANUAL

MODEL 2001-8

PET 2001-8

PERSONAL COMPUTER

USER MANUAL

OCTOBER 1978

first edition

Commodore Business Machines
901 California Avenue
Palo Alto, California 94303

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

Chapter 1. WELCOME TO YOUR PET COMPUTER

Congratulations and welcome to the exciting new world of personal computers. By selecting the PET 2001 you have eliminated the problems of getting a personal computer system running. Your time is now available for learning the functions and capabilities of your PET. In fact, if you follow a few simple procedures outlined in this manual, you should be able to achieve initial operation of your Pet 2001 within a short period after unpacking the shipping container.

The potentials of your PET are virtually limitless. This book, by its very nature, is limited. Questions will arise that this book has not covered or even anticipated.

Write to us at Commodore with your questions. We will answer many that you and other users will pose with a newsletter we'll be sending out from time to time to users.

Commodore Systems Divisions:
901 California Avenue
Palo Alto, California 94304
USA

360 Euston Road
London NWI 3B1
England

3370 Pharmacy Avenue
Agincourt
Ontario, M1W 2k4
Canada

PET is a Personal Electronic Transactor. Everything is complete in one steel cabinet. It contains a CRT board, keyboard, computer board, and a Commodore supplied cassette. There is a built-in black and white television monitor, which displays characters in a format that appear to you to be forty characters by twenty-five lines.

At the heart of your PET 2001 is an MCS 6502 microprocessor. This microprocessor totally controls operation of the screen, keyboard, cassettes, and additional peripherals which can be added to the PET. The product is so construed that you cannot damage the PET from the keyboard. The operating system cannot be destroyed because the computer software, or operating instructions are contained in a fixed memory. (Called Read-Only-Memory) This allows both the first time user and the sophisticated user to use the PET with impunity.

In order to satisfy the needs of the serious user as well as the first time user of a computer product, we have used three formats in this manual.

Summary discussions in this type font are designed to answer the questions of a professional programmer. When you are first using your PET manual, read these sections lightly and spend time on the more detailed explanations which are in the type font of the preceding paragraph. After you have used the PET a bit, the italicized summary sections will be useful when you want to review how a particular instruction works.

The third type of format gives a detailed description of how the PET implements a section. These sections are for people who use the PET at the machine level. The first time reader may find these sections difficult to follow and we recommend he/she use them only on re-reading the material when more familiar with the PET operating system. The language which you will use to communicate with your PET is called BASIC, an acronym for Beginners All-purpose Symbolic Instruction Code.

It should be noted that there are other reference material written about BASIC, with different types of usage in mind, which are available to the public. We have included a list of such suggested reading in the appendix. Some of these manuals may be more useful to satisfy a specific need which is not covered in this particular manual.

The ultimate teaching device for learning how the PET works is the PET itself. In some cases, we will ask you to use the PET along with the text, rather than just continuously reading what to do. In any case, this documentation is sufficient to allow you to get started on an example. By constructing similar examples of your own, any questions which might arise are answered by the machine itself. Of course, we invite comments on this material and particularly on examples which you as an individual have used to resolve any unclear statement.

Lastly, since this manual cannot presume to provide all technical information of hardware or programming aspects of the MCS 6502 microprocessor, we direct you to two other Commodore publications: There is a manual available from your dealer called Hardware Manual which also includes descriptions of the auxilliary devices which generate the control signals necessary to allow the PET to operate. Detailed specifications of the computer and the language in which it operates are available in a book called Programming Manual. This is available for purchase from your PET dealer or either can be purchased from Commodore directly.

NOTES

Chapter 2. UNPACKING YOUR PET AND TURNING IT ON

Please check the carton for any special unpacking instructions and carefully examine your PET for any concealed damage. If anything is amiss, reportly this **immediately** to both the place of purchase and the shipping agent.

Remove your PET from its protective shipping carton and place it on the counter, desk, or other suitable surface, then plug it into any standard, grounded electrical outlet. (In some countries no plug is provided.)

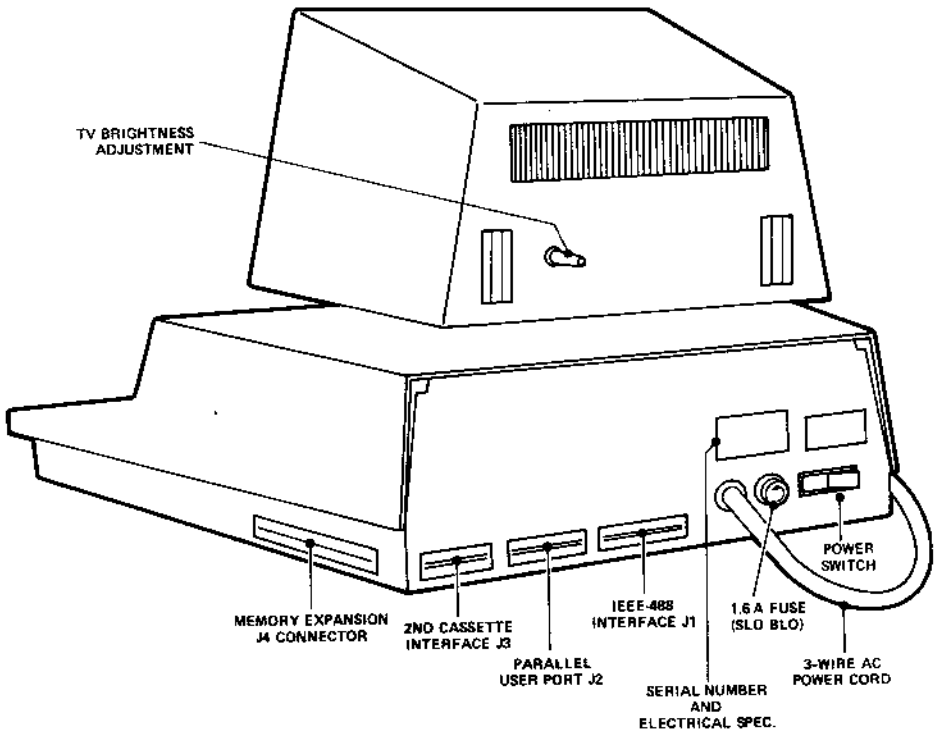


Figure 2.1. Rear view of PET 2001-8 showing switch, fuse, line cord and interfacing connectors

The power switch is located in the left rear of the PET. Closing the switch to the left turns the PET on and closing it to the right turns it off. (There is a white dot on the power switch to indicate it is in the power-on position, or an ON/OFF label.)

Immediately, when the power switch is turned on, power is supplied to the internal circuits. There is a time-out circuit in a special condition (reset condition) which initializes them into a known state. If the screen has had power immediately prior to this time, you will see on the screen a variety of strange characters which reflect the current contents of the computer memory which is controlling the screen. The screen memory transfer to the screen is done with circuitry outside control of the main microprocessor, and so, even when the computer is not operational, the screen always displays the current contents of the screen memory.

At the end of the power-on cycle, the computer initializes the internal memory, blanks the screen temporarily, and then displays on the screen a message like the following:

```
***COMMODORE BASIC***
 7167 BYTES FREE
READY.
□
```

The 7167 refers to available users' programmable memory. A byte is the fundamental data element of the PET computer and corresponds roughly to one letter or digit of information. The 8K model should show- in theory - "8192 bytes". But a few hundred are used by the PET internally. The balance shown "7167" is net available bytes.

If you fail to get the power-up display the first time, try turning the power switch slowly off, then back on.

To get the display, four different types of memory are used: ROM, User Read/Write, I/O (Input/Output), and Screen Memory.
The relationship between these memories is shown in figure 2.2.

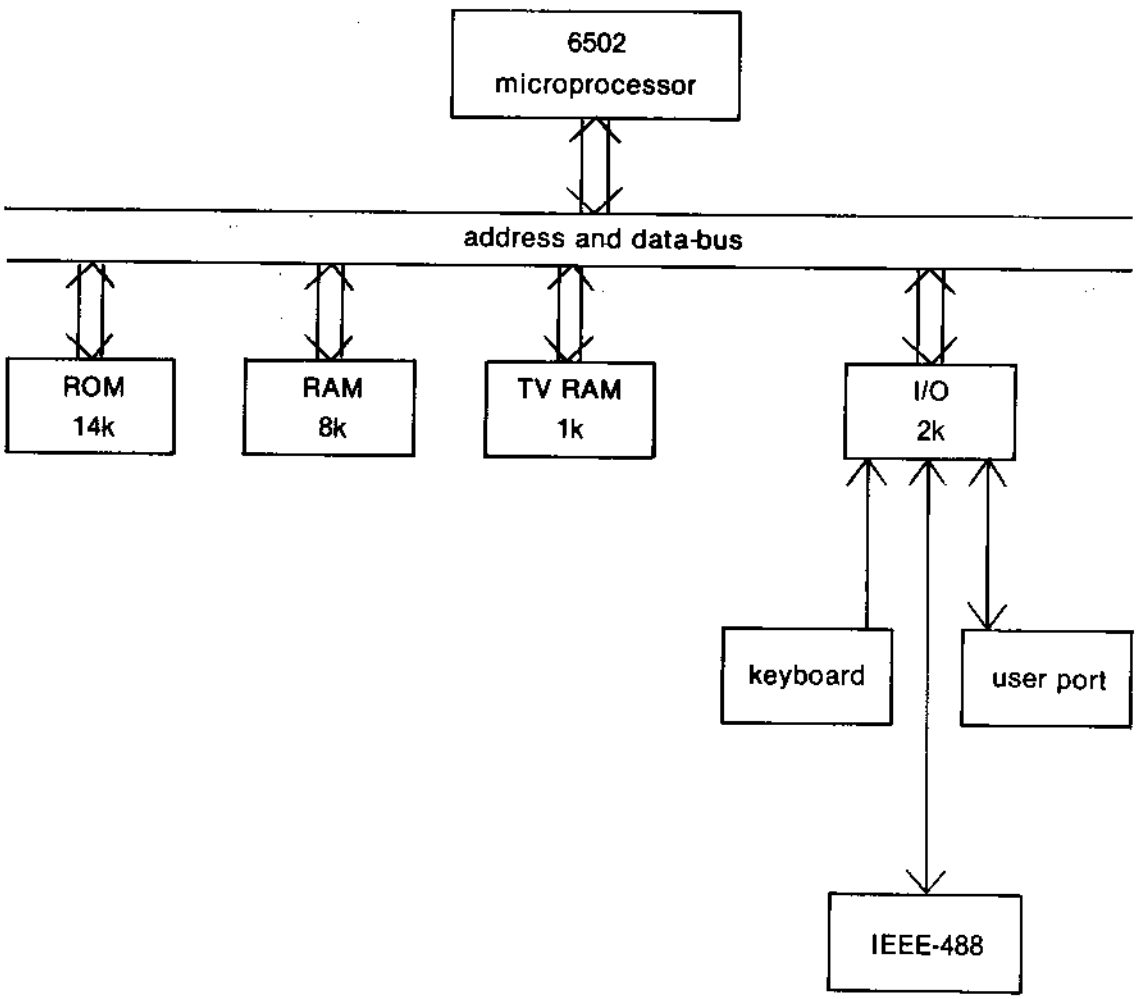


Figure 2.2. PET memory bus

ROM (READ ONLY MEMORY)
ROM causes the PET to perform most of its operations. In each PET, 14K of ROM contains a series of programs written by Commodore which scan the keyboard, print the display, control input/output, count the real time clock, and execute commands that the user types in. Read Only memories are not only the lowest cost memory for storing this data, but also give the user the most protection and the fastest operation of his machine. This is because the operating system memory is indestructible from the keyboard, or from the user's program. Not only is the machine available to run basic from the moment it is powered on, but also the user program cannot damage the basic operating system.

I/O MEMORY
The second type of memory is that which is devoted to Input/output operations. This memory contains I/O devices called PIA* and VIA** which allow the PET to individually control the bits that manipulate the computer. Except when special I/O operations are desired, the user should not allow his program to interfere in any way with these areas. The operating system automatically handles these locations in order to perform legitimate Input/Output operations.

USER READ-WRITE MEMORY - R.A.M.(RANDOM ACCESS MEMORY)
The third type of memory is the User Program Memory Space. (We will call this area RAM throughout this book.) In a standard 8K PET, it is located from location $0000 to hexidecimal $1FFF. A detailed map of all the memory is included in figure 2.3, showing where the ROM, RAM, I/O, and Screen Memory are located from a programming standpoint. As you can see by the map, the first 1024 bytes of memory are reserved for the operating system to use for its various tasks, including the buffering of data from the cassettes and other I/O devices. The message "7167 BYTES FREE" is a result of an analysis of BASIC which starts at locxation 1024 and cycles through the memory to determine which locations are available, thereby, performing a check on whether or not the Read/Write Memory is working correctly.

If the number was less than 7167, you may have a hardware problem. If the number is greater than 7167, you probably have added your own memory. BASIC can automatically check up to 32K of RAM as long as the added memory is continuous to the memory that comes furnished with the PET. This memory is really the working memory in the machine; it is where programs are loaded and BASIC holds all of the program variables.

Later on, we will discuss some techniques to expand this memory by using tape files and program overlays.

SCREEN MEMORY
The screen memory is physically composed of the same kind of chips that are used to make up the PET's standard memory. It is constantly being used by the CRT control electronics, which takes the individual bytes of memory and uses them to address a special character generator ROM, thus displaying characters on the screen.

As mentioned during the power-up discussion, this process is totally automatic, and the programmer has no direct control over it.

---

*PIA - Peripheral Interface Adaptor
**VIA - Versatile Interface Adaptor
For information about these and related chips, see 6502 Hardware Manual.

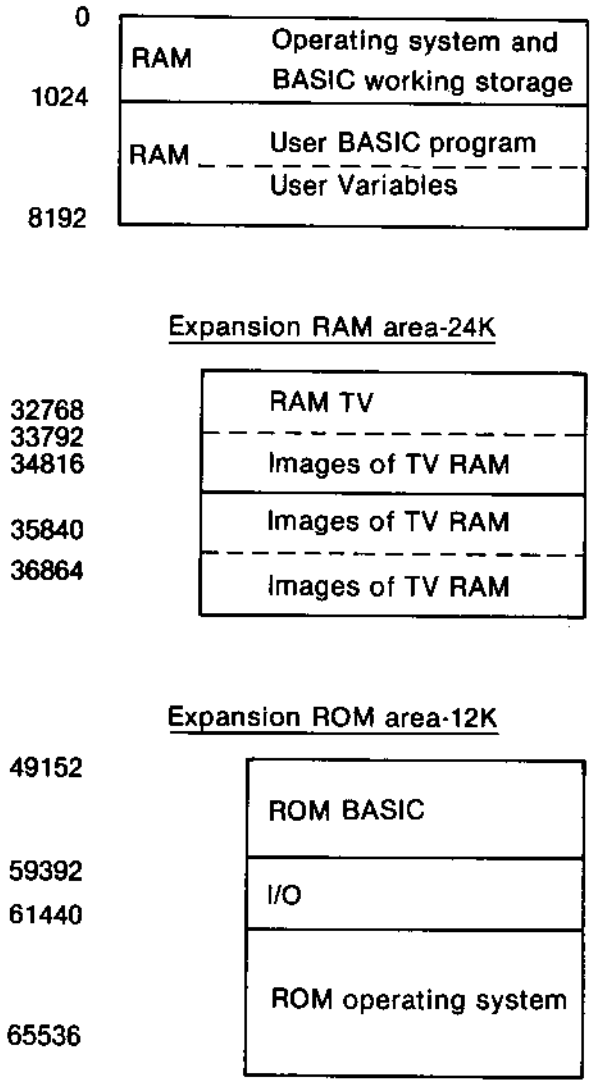


Figure 2.3. PET memory map

On every cycle of the TV screen (1/60 of a second), the hardware starts with the least address ($8000) in the screen memory and processes the screen data starting at the upper left-hand corner of the screen. Each character in the memory is addressed into the character generator eight times, giving us an 8 row high character on the screen. The character ROM that is used generates 8 dots each time it is addressed. These dots are serially fed to the screen, working from left to right and top to bottom. This gives an 8 bit wide 8 bit tall character with no spaces between characters. The CRT controller automatically changes the addressing of the character generating ROM, depending on whether or not it is scanning the top line of a character, the second line of of a character, etc.

There are two character sets stored in the ROM. You can change the character set on the screen by POKEing memory address 59468 with a 14(a 12 turns it back) which turns it to the second character set. After you have played with the screen a little bit, you may want to try this feature to see if your PET performs this way. The second character set substitutes lower case letters for the graphic set that is available in the first set.

To understand this, let us review how characters are represented in the PET and in the memory.

CHARACTER REPRESENTATION IN PET MEMORY

The standard ASCII code is used to represent characters in the main memory. (RAM)

In the PET, the 8th bit (bit 7) is used to signify BASIC command words or graphics characters for the PET screen.

B 6	0	0	0	0	1	1	1	1
1 5	0	0	1	1	0	0	1	1
4	0	1	0	1	0	1	0	1
3210								
0000	NUL	DLE	SP	0	@	P	SP	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	↑	n	~
1111	SI	VS	/	?	O	←	o	

Figure 2.4. ASCII character set (7 bit code)

Example in the PET:

A is represented	0100 0001
Shifted A (a spade) is	1100 0001

The screen memory is organized with a different representation from the main PET memory. There are only 64 characters from the standard ASCII set that are normally printable.

B 5	0	0	0	0
4	0	1	0	1
3210				
0000	@	P		0
0001	A	Q	!	1
0010	B	R	"	2
0011	C	S	#	3
0100	D	T	$	4
0101	E	U	%	5
0110	F	V	&	6
0111	G	W	'	7
1000	H	X	(	8
1001	I	Y	)	9
1010	J	Z	*	:
1011	K	[	+	;
1100	L	\	,	<
1101	M	]	-	=
1110	N	↑	.	>
1111	O	←	/	?

Figure 2.5. ASCII 64 character set (6 bit code)

These are the same characters that are directly available on the PET keyboard.

The representation in screen memory is derived from the standard ASCII set by dropping bit 6; giving us a six bit code for the keyboard characters.

The graphic, or shifted characters, set is represented by a 1 in bit six of the screen memory, giving an additional 64 displayable characters.

This gives the following table for PET displayable characters. It should be noted that all of the graphics characters are organized so that they are just a shift from the normal keyboard character.

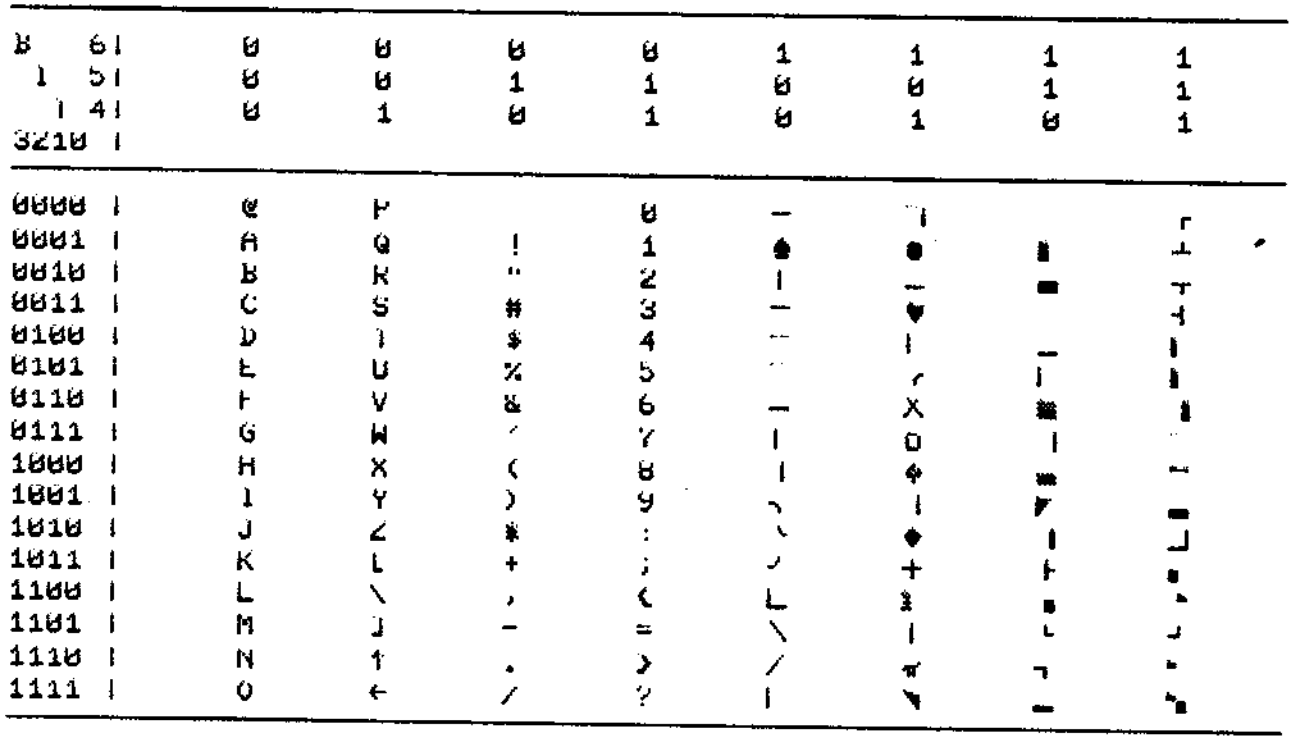


Figure 2.6. PET graphic character set (7 bit code)

Example: This gives us the following conversions:

Character	In main memory	In screen memory
A	0100 0001	00000001
♠	1100 0001	00000001
1	0011 0001	00110001
┴	1011 0001	01110001

Note the reduction from seven bit ASCII to six bit gives the effect of changing the order of A and 1. In screen memory, the 8th bit is used to store reverse field. The reverse field consists of taking the dot pattern from the character generator and reversing it, replacing a white dot with black and a black dot with a white.

If the operating system is used, it automatically translates the values from ASCII into the screen memory representation. Both PRINT and direct input from the keyboard result in automatic translation between the screen memory and the main memory.

USE OF THE SCREEN MEMORY

There are three ways to get data into the screen memory. The first of these is to POKE into the appropriate memory address the desired translated character. This is programmed only when normal updating of the screen is too slow.

As long as the PET directly controls the screen, there is no apparent effect from the fact that the screen and the PET are contending for access to the memory. The routines in the PET change the screen memory only during times when the screen memory is not being used for display. This slows the use of the screen memory down to about 40 percent of the speed obtainable with a POKE. The POKE, however, gives a visual effect of flashing dots, because the screen is displaying the character that is being passed from the PET to the screen memory, rather than the character that should be displayed at that particular position. When a program pokes to the screen, the faster it runs the more flashing there will be.

The second way to get data onto the screen is the keyboard. During a time when keyboard input is enabled, the character being struck on the keyboard is automatically displayed on the screen. The third approach is by use of the PRINT command in BASIC. When

PRINT "ABC"

is typed to BASIC, it results in the next line being printed as:

ABC

This is a print of a literal field in which all characters between the quotes are printed. The next position at which a character will be displayed if typed on the keyboard is indicated by a flashing signal called a cursor. The cursor is a visual indication to the user of the next print position in screen memory.

What is physically happening in the machine is that everytime the screen is recycled, about 1/60th of a second, an interrupt to the PET is generated. This generates a real-time clock on the computer (the PET) and steps a blinker counter. When this counter reads 37, the character referenced by the screen memory pointer is reversed in the 8th bit. This causes the reference character to be shown in alternating normal and reverse field, giving as visual effect of blinking.
By moving the pointer, we can print output any place on the screen. This is done by using a combination of the keyboard and some software called the screen editor, which manipulates screen memory under control of the keyboard.

Whenever the blinking cursor appears on the screen, the computer transfers data from the keyboard to the screen memory.

Keyboard data is transferred by the interrupt routine to the screen memory each time a new key is struck. Only after a carriage return is the keyboard data transferred to the operating program, and then a whole line is transferred at once.

There are two exceptions to this, neither one of which causes the cursor to blink. One of them is the use of GET, which will be discussed in a later section, and the other one is when the keyboard data is accessed directly using machine language programs.

The PET keyboard has been optimized for use as a computer keyboard, though the organization is similar to that of a typewriter so a touch typist does not feel totally out of place.

However, some important changes have been made:

1. Because of the high use of numbers and calculations with the computer, a calculator-like number pad has been added to the right of the main keyboard.
2. The number pad has all of the mathematical operators in a form that is normal for BASIC.
3. The various keys for screen movement and editing are located on the numeric pad.
4. The characters which are normally the shift of the numbers on a standard keyboard no longer require shifting. These characters are quite often used in BASIC, and it is convenient to have them available without shifting.
5. All standard characters are unshifted, so that a complete 64-character graphics set is available by use of the shift keys. These graphics give the PET significant line drawing ability.

PET KEYBOARD

The keyboard consists of 73 keys, including two shift keys, either one of which may be pressed to cause the upper or shifted characters displayed on the keyboard to be operational. Lower characters are always used unless one of the two shift keys is pressed simultaneously. Each key has a thin, transparent plastic film covering the keytop which should be removed. This protection was left in place to protect the keys against scratching during shipping. To remove the film, carefully peel it off by using the sticky side of a piece of masking tape so as to avoid scratching the keytops.

There are 64 printed characters on the keyboard with 64 upper case, or shifted characters on the same keys. The rest of the keyboard consists of function characters. Some of the functions are obvious: like character return or cursor right and left. Reverse on allows all subsequent characters to be diplayed in reverse field - black on white.

The reverse key is operational on a memory basis. From time to time the key is struck, the function is operational until it is terminated by a RETURN pressed or printed, or by pressing reverse-off (the shifted reverse key). This concept of reversal of function, up and down, right and left is carried through to the function keys, so that the complementary functions are usually combined, with one being the shift of the other.

The keyboard is scanned using a 6520 PIA, a four line to ten line decoder and the interrupt routine from the CRT controller. Each time the interrupt occurs from the CRT, the keyboard is scanned using a left to right scan. The keyboard is organized on a 2 x 5 row matrix with the matrix being repeated 8 times across the keyboard. To implement noise protection and N key roll over, the keyboard scan routine keeps the final value of the last scan in a buffer.

Until that key is released, no other keyboard scans are acknowledged unless a later scanned key is struck. The later scanned key is then considered to be the next key closure. The algorithm does not

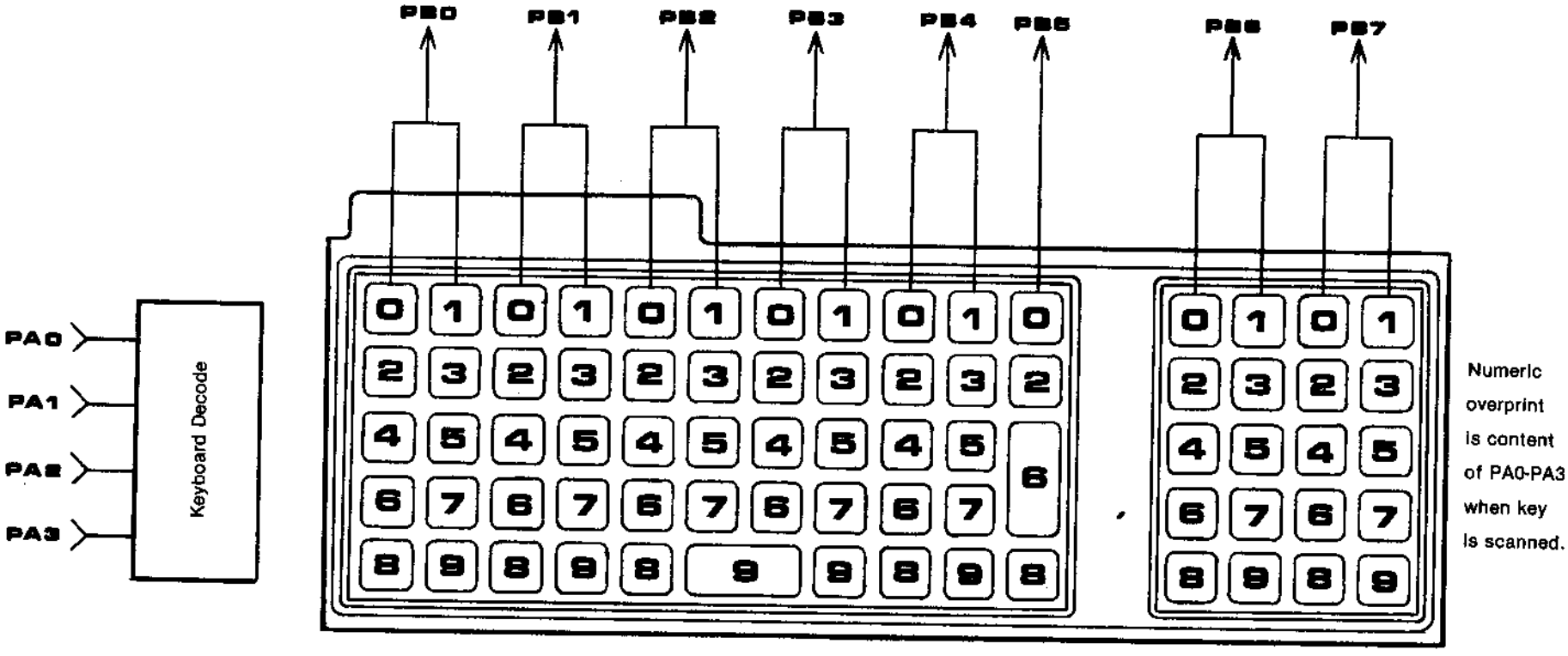


Figure 3.1. PET keyboard scan

PIA Data register addresses PA = 59408 PB = 59410

give classical N key roll over but does allow for legitimate rejection of noise and trapping of the keys in the order that they are struck.

The keyboard is left scanning the last row, which contains the stop key. This allows the routine in BASIC, that checks for the stop key to sample the input I/O device, without having to perform any of the normal functions of scanning. The user can take advantage of this by reading the input character for that row.

The shift key is a special multiple key closure and is treated separately. If either of the two shift keys is pressed, the software sets a special shift switch which is used to change the decode of the key. All key closures are translated using a ROM-based look-up table for the key. The shift key is encoded into bit 8 of the ASCII character which is then translated into the screen representation in the standard way.

Once the hardware translation is done, the encoded value is transferred into a 10 character keyboard queue. The keyboard queue is loaded every time a new key closure is sensed and is unloaded as soon as characters can be transferred to the screen.

This input queue is scanned by the GET routine directly to allow input without going to the screen. The input stack may be scanned by a user program. The user program can look at the pointer at location 525 to determine whether or not it is greater than zero; if it is, that means that there is data in the keyboard queue. The keyboard queue is located at 527-536. The first character may be taken out; all subsequent characters moved down, and a load index pointer decremented by one.

This is a dangerous routine, unless written in a machine language with the interrupt masked, because a new key closure could store a new value during a time that you are scanning and changing the queue. Both the GET and keyboard input routine take care of that automatically by only operating during the interrupt or with the interrupt masked.

Whenever the screen editor routine is operational, a special two-level operating system is in play. The first level enables the cursor to flash and writes data from the keyboard to screen memory at the current cursor position. The routine then moves the cursor one character further down in memory. The process is repeated, trying to keep th keyboard queue empty.

The second level flashes the cursor and translates and stores characters from the keyboard into the keyboard queue. Meanwhile, the first level operating system always watches the input stream for a carriage return. After the carriage return is printed, this routine automatically transfers the entire line to the operating system. The rest of the operating system does not see the characters until they have been typed and a carriage return is sent. This allows for total editing of the line, prior to handing it to the operating system.

An interesting trick for the more advanced programmer is to use the PET to write its own programs. By printing out a line to the screen, forcing a carriage return into the keyboard queue and then returning control to BASIC, new line numbers may be entered into the memory. Another example of the use of the keyboard queue is the LOAD/RUN sequence which is implemented by the keyboard scan program when a shift-run is encountered, the routine automatically forces "LOAD, CARRIAGE RETURN, RUN CARRIAGE RETURN" into the keyboard queue. When control is returned to the input routine, the load followed by the run is automatically transferred in the proper order.

It should be noted that this keyboard queue is only ten characters long and if it is exceeded, dramatically bad effects can happen to your system. The only known recovery from exceeding this queue is to power

the system back on and start over. When fooling with the queue, remember that if the user is typing on the keyboard and you do not have the interrupt turned off, the operating system is going to kill you.

SCREEN EDITOR

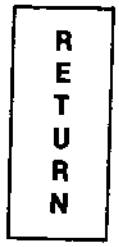
Typing on the keyboard, while the cursor is active, transfers what is typed on the keyboard directly to the screen. This function is like a simple computer terminal which requires you to retype a whole line until you get it right, but the PET lets you edit your mistakes before you enter a line. The editor is best understood with a PET to illustrate it. The user should follow discussions on his own PET, as many of the examples are much more difficult to describe than to see.

To follow these examples, two concepts are necessary. One is that when we type a ? the BASIC operating system is goling to interpret the ? the same as PRINT.

The second concept is that when we follow a ? by a ",all characters after the", until the next " is encountered,
are treated by BASIC as characters that you will want to have printed onto the screen.

In this section you are operating the computer in what is known as a direct mode. (i.E. rather than programming mode). BASIC is executing each instruction like print as soon as you type it into the system and hit carriage return. We will see in the future that this is not the way most programs are operated. It does make the machine useful as a super calculator.

The first thing that we want to do is have the machine type a simple message. You should have already done this with your users' guide. However, we hope by now that you understand a little better. We type the line:

?"HI THERE" 

You will see that BASIC responds by printing HI THERE. It should be noted that each time we struck a key on the keyboard, the cursor moved automatically one place to the right, allowing us to type in the next character, and nothing else happened until after the carriage return. When the carriage return occurred, the HI THERE appeared almost immediately on the screen.

Let us talk about the simplest function; that is, immediately correcting a mistake. Retype the line ?HI THERE B. What we were trying to type was HI THERE PET, but we hit the character B rather than P. For those of you who are touch typists, you may have already made this mistake with the PET's close keys. In order to allow you to immediately correct this mistake, there is a key which allows us to erase a previously struck character. This key is called the delete key, located in the upper right-hand side of the keyboard.

If we strike the delete key once, you will see that the B has disappeared. Typing the P results in an overstrike of that position. We can now finish typing ET; then hit carriage return, causing the PET to print out HI THERE PET, a blank line, and READY.

The delete key is the fundamental editing tool which allows you to strike out as many characters as you want from where you are and then retype. This is the simplest form of editing. It is implemented by

decrementing the screen pointer from where you are by one and striking a blank over where the screen pointer is. We can go back and erase the READY that is right in front of our cursor by just continuously striking the delete key. Notice two facts as you are striking; (1) if you strike slowly, the cursor will move one character at a time, and (2) if you strike fast, the cursor will actually move several characters before you see it blink. This phenomenon occurs because it takes 15 times as long to blink 2 characters as it does to overstrike one. Also, notice that the PET wraps around the screen. The screen memory is organized so that deleting the previous character in memory moves the pointer back over that character. Because of the fact that the characters scan from right to left in 40-column chunks, for example deleting the character at the beginning of the line, and then striking the delete key at the beginning of the line, deletes the 40th character of the previous line. Just keying back 40 strokes erases the READY from the line above, however, this is a pretty slow way of editing.

There are three cursor movement keys on your PET. One key moves the cursor right or left; the second key moves it up and down, and the third key moves it home (upper left-hand corner) and clears the screen.

CURSOR RIGHT AND LEFT

The cursor right key moves the pointer one character to the right. If we strike it now five times, you will see that it moves us five columns over. It accomplishes this by changing the cursor pointer in memory. The cursor left key is on the same key as the cursor right and is evoked by shifting prior to striking. If we type that four times, you will see that now we are back one character to the right of where we started. If we strike it two more times, it moves us around the corner of the previous line. Cursor left, of course, just moves the cursor pointer one character less in memory. Going to the left, it moves one character at a time. Obviously, by doing this, we are able to edit the screen. However, faster editing can often be achieved by use of the cursor up and down keys.

CURSOR UP AND DOWN

The cursor down moves the pointer 40 columns to the right from its current position. This gives it the same visual effect as moving it down one line on the screen. For an example, try spacing over forty positions with the cursor right. The cursor is now on the same position on the screen, but down one line. To cause the cursor to move up, hold down the shift key while striking the cursor up/down key once; this gets us back to our original position.

Cursor up moves the screen memory pointer "up" 40 characters from its current position, or rather, 40 characters less in screen memory than the current position.

SCREEN EDITING

We can now use the cursor movement characters to get up in position on the second H in the HI THERE PET message. Once you are there, you can now delete the T by striking the delete key. You will notice that all the characters to the right of the character being deleted are moved to the left one character. You will now see the delete is actually a matter of moving all the characters in memory left one, rather than just substituting a blank.

INSERT/DELETE

Before analyzing insert and delete, we should be reminded that the screen memory is organized such that any single line may consist of 40 or 80 characters. (See section on screen memory.) Insert and delete are concerned with the characters on a line. Whenever the delete key is struck, all of the characters, starting from the position of the cursor, to the end of the line, are automatically shifted one character to the left, replacing the character preceding the cursor. The cursor is then moved to the position of the replaced character.

The last character in the line is automatically blanked. Insert is the reverse of this process. If we want to

fix the line that we just got through taking the T out of, we need to put a T back between the blank and the HERE. In order to do that, we have to make a space in which to type the T. To accomplish this, we strike the shifted insert key with a single stroke. After striking T, you will note that this now creates a screen which says HI THERE PET, with the cursor blinking over the first character of the insert. To insert more than one character, strike the insert key more than once; this moves all the characters on the line to the right, and the cursor points to the first character of the insert. This then allows us to insert several characters on the line. For example, if we hit the insert key three times, type T's until the cursor is positioned over the H, then delete all of the extra T's; we will then see that the back and forth movement in the line is automatically handled and we end up with a perfectly recomposed message. It should be noted that in no time has the computer responded to these commands, other than making a change on the screen. This is because we have not yet pressed carriage return to tell the PET that the line is complete.

That is why we have been talking about a screen editor. All editing is accomplished between the keyboard and the screen memory, without interfering in any way with the rest of the operating system. This allows the user to compose perfect text and hand it to the computer without the programmer who is using the data, whether it be BASIC or the user program, to worry about the intermediate steps of making corrections. It is best symbolized by:

What You See Is What You Get.

LINES ON A PET SCREEN

Physically, a line on the screen consists of 40 columns of information. However, traditionally in the computer business, many data inputs are organized for 80 column data cards and, of course, much more data can be put into 80 columns than into 40. Therefore, although the PET screen can display only 40 characters per line, the user is given all the flexibility of an 80-column line. This is accomplished by allowing the screen to define more than 40 characters as a line. If we move our cursor over to the beginning of the line below HI THERE, and start typing NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THE PARTY, we will see that after typing the E, the space is automatically on the next line. You will soon see the screen considers this to be an 80-column line although the HI THERE PET right above is only considered to be a 40-column line.

The thing that allows the PET to accomplish this is that internally, there is a table of pointers at the beginning of the line. Each line has a marker that indicates whether it is the beginning of a line or a continuation line. This pointer is kept in the negative bit position of the index pointer. Whenever a cursor up or cursor down occurs, the editor examines the status of these line pointers in order to initialize the PET to their proper line number. At any time while the cursor is on the screen, there is a separate value kept which is the beginning pointer for the first complete line from which the cursor operates. The screen position is then kept as a separate pointer telling the PET whether it is greater or less than 40 characters. Whenever scrolling occurs, the line pointers are moved up in such a way that the concept of the first line second line is maintained until the line disappears on the screen. This line pointer table is located in memory locations 553-577.

Now that we understand that the PET can allow 80 columns, let us see what happens when we do the insert at the beginning. To print this line, we have to put a ?" at the beginning of the characters. We move the cursor up and left, until the cursor blinks on the N of NOW. If we insert twice, we can then type a ?" (it should be noted that this causes the characters on the line to all move to the right). If we now carriage return, the PET prints NOW IS THE TIME on two consecutive lines, spaces a line and types READY. If we

go up and make a change in the middle of the line, we can see that it makes no difference where we hit the carriage return in the line. If we space up to **the** word PARTY the first time that it is on the screen, now even though the cursor is blinking on the P, a carriage return causes the entire line to be reprinted. The basic rule is that when a carriage return is struck, regardless of where it occurs in the line, the entire line is transferred, whether it be a 40-or 80-column line. Sophistication in using the editor will become more apparent as you use it when writing programs.

SCROLLING

Now that we have a mixture of 40-and 80-column lines on the screen; let us investigate what happens when we try to move the cursor off the bottom. To do this, we just cursor down until the cursor is at the base of the screen. Hitting the next cursor down causes the entire screen to move up one line. Any time we attempt to print past the thousandth character on the screen, the screen editor automatically moves the entire screen up one line.

Lines move up on the screen by a one line or two line jump depending on the status of the top line on the screen. This is accomplished in hardware by checking the top line pointer plus one. If an 80-column line is to be scrolled off the top, the 81st character through to the thousanth character are moved to the top of the screen memory, and the bottom 80 characters of memory are filled with blanks. If only a 40-column line is to be moved off the top, the 41st character is moved to the first, etc., and 40 characters are blanked at the bottom of memory. The cursor is positioned automatically in the same position at the bottom of the screen as it was when you tried to move the cursor down; or in the case of a carriage return and/or printing, the cursor is moved automatically to the left-hand side of the bottom line.

This process is totally automatic and is caused by attempting to print carriage return or space off the bottom of the screen. There is no other program control over the movement. As we will see when we write a program that causes scrolling, the scrolling speed on the PET is too fast to read. If the reverse key is held down while printing is occurring, the scrolling will be slower by a factor of 20.

HOME AND CLEAR

Striking the home key moves the cursor to the upper left-hand corner of the screen (the first location of the screen memory). Holding the shift key down and pressing the clear key gives you a blank screen with the cursor blinking in the upper left-hand corner. This is accomplished by moving blanks into all thousand screen positions and again homing the cursor. Clear or home can be given at any place on the screen.

The PET basically moves data from the keyboard to the screen and then when a carriage return is struck moves the screen data into a program. This allows the user the flexibility of making a correction on the screen without having any effect on the program that is going to receive the corrected version. Keys are provided to allow movement around the screen and to insert or delete, as well as type over any character on the screen. This allows the entire screen to act as an editing place for user-controlled input.

The combination of instructions to solve a particular problem cannot be taught in a text book. It is a creative process. Someone who knows how to use the computer uses his intuition or careful planning to figure out instruction sequences to allow solution of his problem. All that we can cover in this book and all the PET can be - except when it is provided with pre-programmed softrware - is a tool to use for solving problems. This book cannot teach you to solve your particular problem. It can, however, teach you how to use the PET as an instrument.

THE PRINT STATEMENT

A computer can calculate numbers all day but it is of no value unless the computations can be displayed. We will begin our discussion of BASIC with the PRINT statement for that reason.

When typing text, PRINT can be abbreviated as ?. A statement such as this:

PRINT "HELLO"

is an instruction to the computer telling it to display on the screen all characters between the quotes -- in this case a word of greeting. On the other hand:

PRINT 1024 *8

is an instruction to print the product of 1024 multiplied *8.

It is useful to note that BASIC allows you to print more than one value at a given time. Rather than having it write a line, print 'A' and on a second line print 'B', it is possible to write the line:

PRINT 1024↑ 2, 1024 ↑ 3

which will print the square of 1024, a few spaces, and then the cube of 1024. Details of the exact format is contained in the next section. The point here is that you can print as many values across a series of lines as you can write down.

Unless the computer has been instructed otherwise by means of CMD command, all print outputs are directed to the built-in screen. The characters are printed in the next available print position on the screen, under the control of BASIC and an editor which is keeping track of the screen position. Although the physical representation on the screen is 25 lines by 40 characters, the printing of up to 80 characters is accomplished by the screen automatically folding over the 41st character onto the next line. The computer automatically scrolls the screen up one or two full lines when it reaches the one-thousandth character on the screen.

The command PRINT has two major forms under the control of BASIC. (1) The standard print single character which allows for printing the field specified after the print statement has ended in the form print variable. If the data is presented in this form, the field is printed starting at the current screen position and followed by a carriage return. (2) Data presented in the form PRINT A, B, then BASIC automatically tabulates printing 'A' starting at the current screen position then spacing over 10 characters, prints 'B' followed by a carriage return. In order to cause BASIC to not send the carriage return after B, a ; (semicolon) is used. PRINT A;B; results in the 'A' being printed, then followed by no extra spaces, variable 'B' is printed. The cursor is left at the end of the 'B' field. If the variable A is more than seven characters, 'B' will be printed after spating 20 characters,when using PRINT A,B.

BASIC obeys the following roles for printing characters. When the field to be printed is a string, there are no leading or trailing characters sent. If the field to be printed is a number, BASIC first checks its size. If the number is less than .01 or greater than or equal to 999999999.2, BASIC prints it using scientific notation. For example, .0034 is printed as 3.4 E-03 and -1234567890.5 is printed as -1.2345678E+09. If the number falls between these values, the most significant 9 digits are printed, plus a decimal point if

needed. Trailing zeroes after the decimal point are not printed. BASIC always prints a skip character after a number (unless it is printed as a string).

It should be noted that in order to take full advantage of the PET's ability to compose text material on the screen, unlike most BASICs, the apparent space between fields is always a skip (cursor right) character in the PET, which causes the screen to advance the screen pointer by one character; it does not result in any of the data screen being covered.

Because the PET allows the inclusion of all cursor positioning as literal characters within a string, the programmer has full control of the screen print position. The cursor control characters available to use as literals are clear screen, home cursor, cursor right, left, up and down. By use of these literals, one can compose fields of any length and in any size starting in any one of the 1,000 character positions displayable on the PET screen.

We previously discussed how the PET screen memory consists of a thousand characters of storage located at memory location 8000 hexa-decimal. Characters are represented in screen memory in six bit ASCII code, concatenated with two additional bits. One of these bits is a reverse field and the second one is the upper-lower case bit.

When printing to the screen, the print subroutine in the operating system automatically translates ASCII characters into the screen memory form. The various screen control characters are simply movement characters for the screen printer. The home character moves the printer pointer to the beginning of the screen. The clear character moves the printer pointer to the beginning of the screen, and inserts the representation for blank in all of the 1000 characters on the screen.

In BASIC, numbers are represented as 5-byte binary quantities, except in the special case of integers, which are represented in two bytes. As far as printing is concerned, BASIC prints integers the same as it does floating point numbers. In fact, BASIC automatically converts integers to floating point and then the floating point print routine converts the floating point numbers into printable characters.

VARIABLES

We have already seen that the PET can be used as a large calculator which performs mathematical functions and then can print the results. However, in many cases, programming consists of developing intermediate values or performing operations until something equals a certain value. In order to implement programming at any level, we need to establish the use of functions which can have a variety of values at any one time. A function that can have any value is defined in both algebra and in programming as a variable. If you are not familiar with the concept of a variable through mathematics; then a book on beginning algebra, or perhaps one of the very rudimentary texts on BASIC might help you. All of our discussions after this will concern themselves with the use of variables.

In BASIC, variables are defined by two character alpha numerics. If the variable is a numeric variable then it has no trailing character. The character A is considered to be the variable A. Characters AA is a different variable. Characters A1 is a third variable, but all three are defined as numeric values. If the variable contains alphanumeric data, it is defined as a string. A string variable now ends with a $. Thus, A and A$ are numeric and string values respectively and are different variables. AA$, likewise, is different from AA, etc. BASIC distinguishes a variable by the fact that the first character is always an alphabetic character. The second character may be either numeric or alphabetic. An integer variable ends with %, e.g.A%.

ARRAYS

Arrays is the fourth type of variable which can be defined in BASIC. Arrays are differentiated by the

parentheses which follow them. Parentheses define the particular value within an array which is to be used in an expression.

A(0,1) refers to the first character in the second row of a two-column array and is different from A, A$ and A%. All may be specified in the same program. Specific definitions and memory allocation techniques for each of the types of variables follows, but first let us address some examples of how one uses a variable.

Equal is used in two ways: If encountered in an IF-THEN type of statement, equal means the standard mathematical function: the value to the left of the expression is compared and must equal the value of the right. Otherwise, when following a variable such as in the expression A = 2 + 2, = means replace the value in A with the resultant of the expression to the right.

Originally BASIC required the word LET before any variable assignment, but in PET the LET is optional and may be omitted. A = 2 is equivalent to LET A = 2. The command CLR sets all variables in PET to zero. To understand how variables operate in BASIC, try the following examples on your PET. Remember to press RETURN after each command you enter.

```
CLR
?A
```

PET prints 0.

Now type

```
A=2+2
?A
```

This time PET prints 4.

Now type

```
?B
```

PET prints 0.

Now replace the value of B with twice the value in A, by typing

```
B=2*A
?B
```

PET prints 8.

Now change the value of A by typing

```
A=2+3
?A
```

PET prints 5. If you now type

```
?B
```

PET prints 8, the same value as before. Until we give a new expression for B or re-execute the one which says B = 2*A, the value of B will remain 8.

FLOATING POINT VARIABLES

BASIC always assumes operation, or operates totally, in floating point arithmetic. Therefore, each normal variable is assigned space in memory for a standard floating point number.

Four bytes contain a binary representation of that precision. It gives us the capability of specifying about 9 digits precision of a decimal number. Accuracy of most calculations is limited to this representation. Each variable is also assigned a 1-byte exponent limited to having a maximum value of +33. Exponents less thgan −34 yield numbers too small to distinguish from zero.

STRING VARIABLES

A string variable can contain a function, whether it be a number, graphics character, or standard ASCII character. There is a specific set of variables that allow extraction and packing of data into strings which

will be discussed later on. The string is limited to the 80 characters of the input buffer. There is a specific set of functions that allow the construction of strings up to 255 characters (see later text).

INTEGERS

As we have indicated, an integer is simply a whole number. Floating point variables are stored in BASIC with five bytes; one for the exponent and four for the mantissa, which gives an accuracy of 9 digits. In many cases, variables can be expressed in much simpler numbers. In order to allow the user most memory efficiency, particularly in the case of arrays which can take significant amounts of memory, the PET has implemented the concept of storing certain numbers as two-byte integer values. Any integer value between minus 32,767 to plus 32,767 may be stored in the form of a two-byte number with the highest bit of the number containing the sign.

USE OF PROGRAM AND DIRECT STATEMENTS

Throughout the text, until now, we have been using the program technique which allowed us to get the PET to respond directly to the print statement. In this case, BASIC is obeying the command we are giving it directly, as we type it from the keyboard and hit carriage return. This is so-called direct mode. In this mode, we can use the PET as a super calculator. For instance, if we want the PET to add two numbers and divide the result by a third, we can ask it the question ?(2 + 8)/5. If you have typed that on the PET, you should get the answer of 2 followed by a READY. The PET will obey any statement given it from the keyboard, except when it is in the process of executing a BASIC program. In addition to using it as a super calculator and for teaching with the PET, the direct mode is quite useful for debugging of computer programs. Variables can be assigned intermediate values and then small sections of the program can be executed with GOTO statements to assess why any particular piece of code is not working correctly. Break points can be put in programs and current status of variables checked with print commands, again in direct mode, without having to modify your main program. However, except for debugging or in the case of using the PET as a super calculator, in order to get the computer to act as a true computing element, one has to write or load a BASIC program. The difference between execution in direct mode and a program is that several statements can be grouped together in logical order and then BASIC will execute all of the statements before asking the user for control.

Suppose we want BASIC to print our HI THERE message vertically as opposed to horizontally. We can easily accomplish this in a program but not very easily in a direct statement. Rules for program entry are very simple. Any statement you want to be treated by BASIC as a program statement must be preceded by a line number. A line number may be any number from 0 to 63,999.

A good habit to develop when typing in lines of a program is to use increnments of 10 or 100. Instead of 1, 2, 3, etc., use 10, 20, 30. This will give you space later to add lines and make corrections in your program. All you need to remember is that BASIC interprets each line number in order.

To print HI THERE, vertically, each line of our program will type one letter of the message. we are going to start with line 10 and make each line a multiple of 10.

```
10?"H"
20?"I"
30?"T"
40?"H"
50?"E"
60?"R"
70?"E"
```

Whether you are typing in a program or giving direct commands like RUN, you have got to hit RETURN to tell the PET to take a look at what you have typed and act accordingly. The lines ten through seventy

constitute a program which tells the PET to print out HI THERE.
The program is now resident in memory. To execute the program, type RUN. This gives us the HI THERE printed in the vertical format:

```
H
I
T
H
E
R
E
```

You will note that we do not have a space between the I and T. One of the reasons we use the numbers in the multiple of ten is that we can now insert a correction between lines 20 and 30. First, display the program by typing LIST. This gives us the program printed as follows:

```
10? PRINT "H"
20? PRINT "I"
30? PRINT "T"
40? PRINT "H"
50? PRINT "E"
60? PRINT "R"
70? PRINT "E"
```

Now type:
25?""
Press return and relist the program, and we will see that line 25 is inserted between lines 20 and 30. If we run the program now, we get:

```
H
I

T
H
E
R
E
```

This example demonstrates the use of line numbers and the ability to insert lines numbers to make a correction in a program.

There is another way to get the same effect. First delete the space by typing 25 followed by a carriage return. Then list the program and see that line 25 has been deleted. Now position the cursor on the space following the I on line 20, and insert a cursor down. First by hitting the insert key, and then the cursor down key, if you don't hit the insert key first, the cursor will move down immediately. But because you inserted the cursor-down (it looks like a reverse field Q), the cursor will not move until instruction 20 is executed. Do not forget to hold down shift before striking insert.

When we now run the program, you see this also gives you the effect if a space on the next line. This would not always be true, except we had been cheating and using the automatic scrolling capability of the PET which clears out the field. Had we programmed a home prior to printing a program, we would not have received such a nice result. Try programming a home 5?"HOME", then try a clear 5?"CLEAR".

The screen editor will allow you to take a program and make changes on any of the lines you display on the screen. The list command has several features to help you get the right lines to the screen to edit. List takes programs and prints the contents of the basic program which is stored in memory. The command L-I-S-T starts at the first line number in memory and lists to the screen device all the instructions to the

end. The longer programs features of list which allow you to list only a single line number LIST 20 which lists just line 20, LIST 10-50 which lists lines 10 through 50 included, LIST-50 which means list all the numbers from the beginning of the program through line 50 included, and LIST 50- which lists all of the lines from line 50 to the end of the program. Some combination of the above can be used to find and correct any piece of program which is currently stored in memory. Try each of the above commands on your PET just to see what they do with our little program.

BASIC is an interpretive language related to the direct commands we are executing. BASIC executes a command by taking the last line typed to it and analyzing the line working from left to right looking for key words and expressions which it recognizes. Every time it encounters a key word such as PRINT (or ? which is the token for PRINT), it interprets this word into a command which means something to BASIC. Command words are stored in memory with bit 8 on to tell BASIC that it is a command word, or key word. As a program line is entered into RAM memory through the use of the carriage return, BASIC takes the line number and searches through memory, until it finds the same number, or the number just greater. If it is the same line number, then the entire line in memory is deleted and a new line is inserted in memory. In the pre-interpreted state all the key words are replaced with the single character token for the key word. This allows the interpreter to store commands in the most memory-efficient form. The only data stored is the data typed in by the programmer such as literals, pointers to the variables, and the keywords. PRINT, even though it takes five characters to type, only takes one character in memory.

BASIC is called an interpreter because the actual execution of the instructions is done by analyzing the keyword that needs to be executed in the program line, then executing that keyword under the control of a series of subroutines. This is a trade-off which results in very memory-efficient storage programs but longer execution times than would be true of a machine language program. Because PET BASIC uses tokens in memory and stores them on I/O devices whenever a program is loaded and saved, the actual coding of data on tape or in memory is not transferable to other machines. It is generally not possible to use BASIC instructions typed in from other machines.

When you create a BASIC program you are operating under two levels of editor: the screen character editor and the BASIC line editor. The screen editor allows you to change characters within a line until the carriage return transfers it to main memory. The BASIC line editor allows you to add new lines and modify and delete old lines.
To delete a line, you type the line number immediately followed by a carriage return. To modify a line, list it first on the screen and alter it then type a carriage return to re-enter it. To replace a line, enter the same line number with new text and type carriage return.

There are two ways to execute a BASIC program. The first of these is to type RUN. The command RUN first clears all the program variables and initializes the program pointers. Then it executes each instruction of the program in order, starting at the lowest number. Execution continues until there are no more instructions, and END is encountered, or the stop key is pressed. RUN may have as an argument the number of the first instruction to be executed. For example, if you type RUN30, our sample program will print THERE instead of HI THERE. RUN is executed in direct mode. A GOTO statement, also executed in direct mode, operates the same as RUN except that none of the variables are re-initialized. The GOTO, of course, must specify the line number of the first statement to be executed, e.g. GOTO 30.

LITERALS

In our HI THERE examples we have used PRINT commands with characters to be printed enclosed in quotes. In the PET these are called literal strings. Data is also kept in the PET in binary floating-point

numbers. Much of the data you want to work with in programs is not numeric but alphanumeric -- the way we talk back and forth as human beings.
These characters are specified to the PET with literal strings. More specifically a literal is any value contained within a set of quotes.

To allow the maximum composition of screen data, the PET has a special set of graphics characters and the ability to store and execute cursor control characters which are fed to it by means of literals or other more sophisticated techniques.

We have already discussed in a section on PET keyboard input how the PET stores its data in ASCII. Graphics characters are stored as an extension to this set. Graphics are produced by shifting from the original 64 character set and they are stored in memory with a special indicator to differentiate them from the lower characters on the keys. A literal can be used to draw a line just as easily as it can be used to print HI THERE.

Any combination of characters within the PET keyboard may be typed in as a literal and this includes all cursor movement and the reverse field. PET has a special mode in the screen editor which assumes that you are typing in a literal whenever a quotation mark is typed. From the time that the first quotation mark is typed until the time that a closing quotation mark is entered, all characters are transferred directly to the screen in a format so that the software which transfers the input line to BASIC will transfer them as control characters if that is appropriate.

You can see the cursor movement characters flagged with reverse field within a literal. Type a single quote and see this happen. Reverse field looks like an "R". Home is an "S" and clear is a shifted "S" or heart. Cursor down is a "Q" and cursor up is the shifted "Q" or hole character. Cursor right is a right bracket and cursor left is the shift of that character and looks like a vertical line through the 5th column of dots. Insert is a shifted "T" which looks like a second vertical line.

You cannot enter a character in reverse field into a literal but you can turn on reverse field with the control character before your character is printed. The only characters that are allowed to appear in reverse field between quotes are those which are interpretted as control characters.

Delete is the only editing character that will still work within a literal. Once an odd number of quotes has been typed on a line, you lose the ability to move the cursor about the screen until either a closing quote or a carriage return is typed.

You should note at least one time while you are editing that you have fallen into the aforementioned trap of trying to move the cursor after a quote has been typed. Either type a phoney closing quote or a carriage return, then cursor up to edit your mistake.

Another method of inserting cursor control characters into already existing text is to use the insert function. It has the same effect as an opening quote. For example, if you type insert three times and then try to do a cursor movement, the control characters will be flagged with reverse field just as before. This mode is easy to get out of because you need only enter as many new characters as the number of times you struck the insert function. It is suggested that you make up your own examples to play with this. Examples may also be suggested to you as you make a few editing mistakes.

The ability to readily manipulate the graphics and the cursor movement characters can allow whatever depth of graphical capability you have the time and patience to program. The computer should be fun. We recommend that you develop your own programming skills with the text and contionually experiment with the use of imbedded graphics and cursor movement characters. Remember that you cannot hurt the

machine - the worst that can happen is that you clear the screen accidently after typing in a bunch of stuff.

REVERSE FIELD

We have shown in the examples of quote mode and insert how once a mode has been established for a line, the PET will continue with that function until it is either cancelled by a new control character or a carriage return. Reverse field works in the same way. It remains in effect until a reverse field off character is typed or a carriage return is entered.

As described in a previous section on screen memory, reverse field characters are stored with a special bit on to indicate the black spots in the characters coming from ROM will be all white and all the white spots will be black. As you will see when you type an example, this gives a very desirable highlighting effect and doubles the number of potential characters which the PET can display. This feature is so useful that it is not only implemented on the PET display but in some of the PET hard copy printers as well.

Here is an example of how reverse field works: Clear the screen and type HI (space). Next hit reverse field on and type THERE .Finally type reverse field off, (shifted reverse field on), type (space), PET . This gives us a line in which we have highlighted THERE.

Reverse field remains on from the first time the control character is typed and all characters subsequently typed on the screen will be printed in reverse field until the mode is terminated as we previously mentioned. This applies equally to keyboard input as well as characters printed from a literal string.

To get the PET to type out in reverse field we use a literal with the control character for reverse-field-on inserted. TYPE ?"HI (reverse field on) THERE (reverse field off) PET". Note that the reverse field on and off characters occupy a space on the screen when programming and that they appear in reverse field, but the THERE is not in reverse field yet. The effect of the quote is to postpone the action of a control character until the literal is interpreted. Since the reverse field is turned on by setting a bit of each character in screen memory, a screen position is not required for reverse field on or off when the stream of characters is received by the program which prints it on the screen. Reverse field remains on until a reverse field off character or a carriage return is typed.

TERMS AND OPERATORS

The communication with BASIC is either with numbers or with alphanumeric literals. Numbers are always presented in decimal form even though the microprocessor in the PET operates in binary mode. In order to keep the two straight, PET will assume that whenever we are talking about a number, we are representing it in decimal form. Later when we talk about hexadecimal numbers, they will always be preceded by a $--e.g. $00 10 is equal to 16.

As BASIC recieves lines, the interpreter divides the characters it sees into several classes. Such as commands, functions and operators. PRINT is a command to BASIC with a specific function that PET can perform.

A function can be something like square root or a variable, or a special function. Whenever you type π on the keyboard, you get a constant of 3.14159265, which can be used in an expression.

An operator is a character that is interpreted by BASIC as an arithmetic function which is to be performed in evaluating an expression. The following set of operators are defined for BASIC:

Plus sign (+) causes two values to be added together using floating point representation with the results

being calculated in a floating point accumulator. The accuracy is limited to 9 significant digits. Minus subtracts the value to the right of the minus from the value to the left of the minus sign.

* is the BASIC multiply. The value to the right of the multiply is multiplied by the value to the left.

/ is BASIC's divide. All the numbers to the right of the slash are divided into the expression to the left of the slash.

↑ means exponentiation. All the values to the left of ↑ are raised to power of the value on the right.

Open and close parentheses cause values inside them to be single expressions. All expressions inside parentheses are evaluated as a single value. Parentheses may be nested and are evaluated outward, starting from the innermost set of parentheses. In order of precedence, the memory aid "My Dear Aunt Sally" will help you remember the precedence of operators Multiplication first, then Division, Addition, Subtration. Expressions within parentheses are evaluated first starting from the innermost set of parentheses. The following set of examples should be tried on your PET to show the operation of the operators and their precedence.

Addition
? 2 + 2

Subtraction
? 4 – 2

Multiplication
? 6*2

Division
? 12/2

Use of Parenthesis
? 4 + 8/2
? (4 + 8)/2

Order of Operations
? (2 + 4*(8 – 4)/2)*3

FUNCTIONS

There are three functions which are available in BASIC which are, at the time of writing, unique to the PET. The first of these is TT: Whenever this character is used in an expression, BASIC translates it from the keyboard character of TT to the value of 3.14159265 etc. It can be used anywhere in any expression and will always be evaluated as this number. Example: ? TT.

TI$ and the value TI are two ways to communicate with the real time clock within BASIC. As previously indicated, every time a screen refresh occurs, (1/60th of a second), a value within the PET is updated. This value is measured as a 24-hour real-time clock. It is available to the programmer in its binary form by the expression TI, which gives the value the current number that BASIC is keeping. This number is kept as a three byte binary number whose value is stated in numbers of 60ths of a second, or so called jiffies. To evaluate the amount of time that a particular operation has taken, TI can be stored in a variable at the beginning of the sequence and then the difference calculated by subtracting that variable from the TI at the end. This function is accurate to 1/60 of a second.

TI$ presents and accepts data in the form of hours, minutes,and seconds. When the expression TI$ is used, it always presents data in string form with two characters for hours, two characters for minutes, and two characters for seconds. The value of time in the computer is kept in a 24-hour clock. If it is ten

minutes past 1 p.m. in the afternoon, TI$ would be printed as 131000. To set the value of the real time clock, type the expression TI$ = with the number being typed in quotes in 24-hour time. For example, to set the clock to 2:45 and 30 seconds in the afternoon, type TI$ = "144530".

As a personal experience, you should set the value TI$ = to the right time now and after you have done some additional reading, go back and print it. As with all the other variables, the power-on sequence to the computer zeros the real time clock.

Care must be taken in use of the value TI. Remember that the expression TI automatically goes back to zero at midnight. One of the authors wrote a loop in a program for graphics display where the program is waited until the variable TI is greater than a constant and the value of TI when the display is put on the screen. This expression never reached the computed value as TI goes through midnight. The only way to compensate for this is to watch for when the time might go through midnight, and readjust the stored value when it might.

Functions are preprogrammed capabilities of BASIC which can be treated as a single value. Functions range anywhere from π, which i s a predefined function, to sine, which is a capability of BASIC to compute the sine of a number. When BASIC encounters the code for function, it evaluates the expression for the function, calculates the resulting value, and uses the value in the command. The use is really quite simple. If A equals sine of π radians, the expression would be written:

A = SIN(π)

In this statement, we are actually using two functions, π, and sine; BASIC would evaluate this expression by expanding the value of π, evaluating the function sine and finally storing the result in the variable space for A. In the expression:

A = 2*SIN(π)

after the sine is computed, it is multiplied by 2 and stored in A.

The trigonometric functions, sine, cosine, tangent and arc tangent are all available in PET BASIC. The expressions for SIN, COS, TAN all have as their only argument an angle given in radians. To convert from degrees to radians, multiply the number of degrees by π/180. For example:

?SIN(90*π/180)

calculates Sin of 90 degree. To obtain the cosine of 45 degrees:

PRINT COS (45*π/180)

To compute the tangent of 40 degrees. For example:

?TAN (40*π/180)

Each of these functions are computed by tables. Because π is limited to 9 significant digits, in general, values should be less than 1000 degrees or 6π.

The accuracy of BASIC functions is five parts in ten to the tenth as long as the argument is below 20 radians. Expressions which use the values in radians are a function of the value of π which is accurate only to ten to the ninth. Arc tangent is the only inverse trigonometric function specified as a function in BASIC. The function arc tangent computes the value in radians of the expression given on the argument. Answers are always given between plus or minus 17. The accuracy is 5 parts in 10^{10}. In normal use the result is in radians.

?ATN(.5)

To convert the number to degrees use the following example:

?180/π*ATN(.5)

The following general expressions can be used to compute the value of arc sine and arc cosine as a function of arc tangent.

ARC SIN (X) = ATN(X/SQR(− X*X + 1)

ARC COS (X) = − ATN(X/SQR(− X*X + 1) + 1.5708

Both the above expressions give the results in radians to be converted to degrees by multiplying the total expression by 180/π. (It should be noted that in both the expressions there is a possibility of performing a division by zero which will result in a basic error. Before using the expression, the arc cosine should be checked for zero and before using the expression arc sine, X should be checked for it being equal to the value of one.

MATHEMATICAL FUNCTIONS

The largest legal number that BASIC can handle is ± 1.70141183 E + 38. Any larger number gives an ?overflow error. The smallest magnitude that can be distinguished from 0 is 2.93873588 E − 39. Any smaller
number will result in an underflow.

ABS

Absolute value is specified in the form ABS(X). The function returns the value of the expression as a positive number. There is no inherent accuracy loss. For example:

PRINT ABS(− 145).

INT

This expression basically rounds the current value of the parameter to the next lowest integer. For example:

INT(.23) = 0
INT(− 2.5) = − 3
INT(1.79) = 1

Other than the inherent inaccuracy of dropping significant digits, this expression introduces no additional inaccuracy. However, small inaccuracies in the argument could cause problems. For example, the number four might, in fact, be stored in BASIC as 3.99999999. When this number is used in the argument for an integer, the result is 3, not 4.

SGN

This expression returns a 1 if the sign of the number is greater than zero, a zero if the value is zero, and a − 1 if the sign is negative. For example:

```
?SGN( − 45)
 − 1
?SGN( + 10)
 1
```

SQR

This function calculates the square root of any number greater than zero. If a minus number is used, the result is an ?ILLEGAL QUANTITY ERROR. Accuracy of the expression is 5 parts in 10 to the tenth for the entire range.

?SQR(16)

The following two functions send themselves with natural algorithms. The algorithms are base E which is 2.71828183.

EXPONENT

The parameter defines the power to which the base E is raised. The limit of the parameter is 88.02969189.

A number greater than that will result in an overflow. A form of the expression is EXP(X). Although the PET only allows the flow function for E, other functions are available by ratioing to the Log:

?EXP(1)

Basic logrithmic function is given with the parameter LOG(X) which is logged to base E.
To calculate the LOG to base 10, the expression is written:

LOG(X)/LOG(10)

RANDOM
The random functions are useful for many statistical programs and games. Basic random functions are provided. The random number generator uses an algorithm which develops a value between zero and one. The argument can be either non-zero, or negative. Positive numbers always return the next value of the random number sequence generated by a numerical algorithm in BASIC. It always starts with the same value, or seed power-on. However, the seed for the random can be initialized by using the minus value. Repetitive access to the random function in a program is not random because the relationship of the time is predictable from the time that the program is initialized. So in a fixed program sequence, the only truly random number is the first one. A solution to this is to use the time to generate random seed, use the RND(−TI) to seed a number sequence, and use RND(+1) for the numbers in the sequence. This should give a close to theorhetically pure random number for statistical analysis and definitely gives a random sequence for game play.

The RND of a minus number is not truly random at all. The parameter is passed as a seed to the random number generation sequence. This technique can be used in debugging programs in a sense that a predictable repeatable sequence can be obtained by RND minus for program development. Suppose in a game program you want to simulate rolling a six-headed die. Initially, you can see the random number generator with the instruction

D = RND(−TI)

Subsequently, you can compute the value of the die with

D = INT(6*RND(1) + 1)

PEEK, POKE:
PEEK is a function which allows the user to look at any location in the PET memory. The parameter contains the memory address in decimal in the PET which to want to look at the result is a decimal number between 0 and 255. BASIC is currently constructed so that the contents of any address greater than hexadecimal C000 is automatically returned as zero. This is a legal constraint, posed by the company who wrote the BASIC software to protect their copyright.

Example: To look at memory location 25, the expression is written:

?PEEK(25)

POKE
POKE is not a function but is written like a command. It allows the user to deposit a number into I/O or read/write memory. The parameters are specified in a list after the command. The first parameter is the memory address of where to put the information. It may range from 0 to 65536. The second parameter is the actual value to be deposited. It must be between 0 and 255. For example, if we wanted to put the character A at the first location of the screen memory we would write

POKE 32768,1

Some locations in memory cannot be changed (ROM) and others should not be changed (BASIC and system variable RAM or I/O). If you POKE the latter, be prepared to reset your machine.

USR

The USR is a function which is designed to pass a parameter to a lanuage program using the jump address located at memory location one and two in the PET. See the section on machine language programming for a detailed description and use of this function.

FRE

This function tells you how many bytes are left in memory. Although it is a true function since it can be used in an expression, it is normally used in direct mode in the form:

?FRE(0)

FRE forces a BASIC action called garbage collection. This consolidates all unused bytes into one large block so that they can be efficiently allocated.

Several functions exist to aid in formatting data when it is printed on the screen or hardcopy printer.

TAB

This format function places the cursor at the column specified in the argument. The argument goes through the INT routine. The legal range is $0 \leq X \leq 255$. If the cursor is past the location specified, the tab is ignored. **Note:** TAB uses skip characters, not spaces.

POS

This function returns the position of the cursor. The position is reset to zero at each carriage return. **Note:** HOME and CLEAR do not affect POS even though the cursor is set to the first column.

SPC

This format function prints out the number of skips specified in the argument (which goes through INT). Legal range is $0 \leq X \leq 255$. **Note:** SPC(O) put 256 skips.

NOTES

Use of decision logic in writing programs.

A major advance in BASIC programming is the ability to loop back and re-execute lines of a program. It may be done in two ways -- unconditionally with a GOTO and conditionally with an IF-THEN. GOTO is written to specify a target line number where execution will always branch. GOTO may also be written with a space between GO and TO. PET BASIC will recognize both forms.

```
GO TO 50
GOTO 100
```

IF-THEN has three forms:

```
IF (condition) THEN (statement)
IF (condition) GOTO (line number)
IF (condition) THEN (line number)
```

Conditions are written as two arithmetic expressions separated by a relational operator. PET BASIC provides six relational operators: <, >, =, <>, <=, >=.

Until now we have been developing programs which do single functions in serial order. You should be familiar with the concept that says that first line 10 is executed, then line 20, and other line numbers in ascending order.

If we wanted to take and print numbers betwenn 1 and 20, their square and square root values on the screen, we could write the linear program as before:

```
10 PRINT 1,1,1
20 PRINT 2,2*2, SQR(2)
30 PRINT 3,3*3, SQR(3)
```

The big disadvantage of this is that we would have to keep typing in lines until the 20th line.

```
200 PRINT 20,20*20, SQR(20)
```

UNCONDITIONAL LOOPING

However, with our concepts of variables and the addition of a loop, we can write a program that computes values and prints them out without having to type such a long program.
The program reads as follows:

```
10 PRINT "VALUE","SQUARE", SQUARE ROOT"
```

Line 10 prints a heading for the column of numbers. It is executed only once.

```
20 I=I+1
```

Line 20 computes the next number to use. The first time this line is executed, I has never been referenced so it has an initial value of 0.

```
30 PRINT I,I*I, SQR (I)
```

Line 30 is like lines 10-200 of the previous program except that the constants have been replaced by a variable.

```
40 GOTO 20
```

Line 40 contains a GOTO command which directs execution back to start again at line 20.
BASIC stores text lines so that a pointer to the next line precedes each line. Using this technique, the interpreter can quickly examine only the line numbers, determine if a line does exist, and transfer execution to that line.

GOTO is not limited to branching to a lesser line number but it can branch to a greater number too. You

will see a future example of the concept of using GOTO to skip a portion of code.
As before, we type RUN to start our program. The program will continue to print values of I until the STOP key is pressed. Rapid scrolling of the screen memory makes the screen almost impossible to read, but use of the reverse key slows the scrolling rate. Holding down the reverse key slows the scrolling by a factor of 20.

To stop the loop, press the STOP key. When you want to restart a program either type CONT to cause the program to resume where it left off or RUN to begin at the beginning.

While this program makes use of the GOTO, it does not really help us to solve the problem we tried to address -- printing just 20 numbers on the screen. However, before we address that, let us introduce a small mistake into the program. You should see a common error and its cure. If we retype:

```
40 GOTO 10
```

and then execute, instead of printing a heading at the top of our program. We will intersperse the heading with the computed value. Jumping to the wrong place in the program is the most common error made in programming. Luckily it is most visible in this case. By stopping the program we can use the screen editor to correct line 40 to go to line 20. You have now fixed the first in a long life of program bugs.

CONDITIONAL LOOPING
The IF-THEN statement allows you to specify a case to test and if the case is true, the statement after the THEN is executed. A test is specified by putting one of six relational operators between two expressions.

= equal
<> not equal
> greater than
< less than
>= greater than or equal to
<= less than or equal to

If A<B then print "A LESS THAN B"

If the expression is true, the instructions on the same line with the IF statement are executed. If the expression is false, the program jumps to the next numbered line. If you are in doubt about < and > and what they mean, remember that the arrow points to the value you would like to see less than the other.
In ourexample, we can add the statement:

```
40 IF I <= 20 THEN GOTO 20
```

The IF-THEN lets us make a variety of decisions at the time we are executing the program. This allows us to limit the program and cause actions to happen. In this case, we execute the program from 1 to 20 and then finally drop through the instruction.

We can also write the IF statement to skip around the unconditional GOTO. Add two new lines and restore line 40:

```
35 IF I = 20 GOTO 50
40 GOTO 20
50 END
```

The program will execute through 20 values and when I is equal to 20, go to the END statement.

Most BASIC interpreters required you to include an END statement to finish your program. This is a vestige of when BASIC operated non-interactively from cards. END can be used optionally in PET BASIC to force program execution to end at a specific point.

IF-THEN instructions have three forms: The first is IF expression GOTO line number. The second is IF-THEN line number where GOTO is implied. The third form is IF expression THEN followed by a

statement to be executed before proceeding to the next line. Expressions in this form might change our table to draw a line between the 10th and 11th value on the screen.

```
32 IF I = 10 THEN PRINT "_______"
```

If we try to execute this, you will see that a line is now drawn between the tenth and eleventh value on the screen because of the statement at line 32. It should be noted that the logical conditions of the IF and IF-THEN are only two; either the next line is executed, or the THEN statement is executed. Take care when placing additional programming statements on the line. For example, in:

```
IF X = 5 THEN 50:Z = A
```

the Z would not be executed, because the line either drops through or executes statement 50. However, in

```
IF X = 5 THEN PRINT .X:Z = A
```

the PRINT X and Z = A will be executed if X = 5.

The IF-THEN lets us make a variety of decisions at the time we are executing the program. This allows us to limit the program and cause actions to happen at various points. It is the concept of the unconditional jump plus the concept of testing values that allows the computer to be used as both control element and legitimate computing element. The intelligent combination of logical decisions with repetitive operations makes a program really work.

DATA ENTRY

Before a computer program can perform useful work, it has to be able to access a data base of some sort. The program could require only simple data such as YES or NO responses to a game or simulation. A more complex payroll program might need rates, hours, and tax information. In PET BASIC there are two ways to get information into variables.

READ AND DATA STATEMENTS

Only a short time ago when there were no timeshare systems, BASIC could not accept input other than cards included with the program. Thus, DATA statements were typed and scattered throughout the program. The command READ was designed to pull out this DATA into variables which could be used by the program.

When BASIC began running in an interactive environment through timeshare, verbs such as INPUT and GET allowed direct communication with the BASIC program. READ has been relegated to inputting parameters that change but not as often -- e.g. tables, etc.

The syntax of READ is the verb followed by a list of variables into which the DATA is to be read.

```
READ A, B, C, D
```

READ processes DATA statements as they are encountered in the program. DATA statements at line 10 and 30 might be processed by a READ statement at line 20. DATA is processed sequentially and commas and end of lines are considered terminators

```
10 DATA 2, -53, IE10
20 READ A,B
30 DATA 3.14, 1,06E23
```

Blanks and graphic characters are automatically thrown away unless they are surrounded by quotes. The quotes are considered to be delimiters for literal characters.

String data can be typed without quotes if it does not contain literals.

```
50 DATA ABC, DEF
```

Commas within quotes will not be treated by BASIC as field terminators.

```
60 DATA ",","","","
```

It is also possible to type mixed alphanumeric and data fields. Numeric fields may be treated as alpha.

```
10 DATA 123, ABC, 345
20 READ A, A$, B
```

It is adviseable for the programmer to know how many data statements he has put into the machine or use some kind of a delimeter at the end of the data. If it is not done, the data is continuously read, and the program will index its way through all of the data statements. Finally, DATA will be exhausted and when the next READ is encountered an ?OUT OF DATA ERROR will occur. Sometimes you may also see this error if you carriage return through READY on the screen because the PET thinks you already told it to READ Y.

SYNTAX error results when an attempt to read alpha field into a numeric variable is made.

READ and DATA are implemented in the following manner: The first byte of text contains a zero. This is really not part of the first line but is a dummy line consisting only of a terminator. When RUN is typed, a data statement pointer is directed to this byte. Since it is pointing to a terminator, the first READ command initiates a search for a DATA statement token.

There is one other command available to the programmer which allows him to reuse the stored data. RESTORE restart the DATA search back to the beginning of memory.

The following program would correctly operate continuously re-reading DATA;

```
10 DATA 10, 20, 30, 40, 50, 60, 70
20 I = 1
30 READ A: PRINT A
40 I = I + 1
50 IF I<8 THEN 30
60 RESTORE
70 GO TO 20
```

INPUT

When interactive response to DATA requirements became possible, the concept of INPUT from the keyboard was introduced. Since the classical input device to BASIC was a TTY, the format of input statements was limited by this device.

Operation of INPUT is considerably enhanced when coupled with the powerful PET screen editor. *The form of the statement is the verb INPUT followed by a variable list. INPUT satisfies the variables in sequence.*

```
INPUT A, B, C
```

When BASIC encounters this instruction, it prints a question mark to the screen then activates the screen editor, blinking the cursor for input. Because you are under control of the screen editor, cursor movement characters are allowed up until the carriage return is issued as a terminator.

After carriage return is received, data is handed back to BASIC one character at a time. Data is then interpreted by BASIC using its input buffer and rules of interpretation.

Leading blanks are supressed, so if you are inputting a string which requires blanks or literals, it is necessary to enclose the input characters within quotes.

The editor picks up only the characters between the question mark and the current position of the cursor.

This allows input of data from a pre-constructed form on the screen.

INPUT data may be delimited by commas as with the DATA statement. When more fields are provided than are actually required, BASIC responds with

?EXTRA IGNORED

and takes only those characters it requires to satisfy the INPUT list.

On the other hand, when not enough data is inputted, BASIC will respond with

??

and begin blinking the cursor again to get additional input.

If an alphabetic field is encountered during the interpretation of a numeric field, BASIC responds with a:

?REDO FROM START

In PET, if input is followed by only a carriage return with no other typing, it is considered by BASIC to be a termination of the program, same as a stop key. This particular feature is a carryover from the days of teletype BASIC when this was the most convenient way of terminating a program.
The stop key is not operative while the PET waits for input.

INPUT has a special feature which allows you to indicate to the user what input characters are desired and in what form they are to be. A literal which follows the input command is printed prior to the time the carriage return is typed. For example:

```
10 INPUT "BIRTHDAY"; A
```

it would print:

BIRTHDAY?

and wait for you to input your birthday in standard numeric form to value A. Here is an example of INPUT to calculate the third leg of a right triangle:

```
10 INPUT "FIRST LEG"; A
20 INPUT "SECOND LEG";B
30 IF A=0 OR B=0 THEN 10
40? "THIRD IS"; SQR (A*A+B*B)
50 GOTO 10
```

If you run this program and put in values 3 and 4 respectively, you will get a 5.

We can change our program to se how to combine values on a single line. We delete line 20, list line 10, and change it to:

```
10 INPUT "FIRST LEG, SECOND LEG"; A, B
```

This change, when you execute it, will accept values typed as 3, 4. You will see that either form is acceptable, however, good programming practice protects the user from getting confused as to how many fields go on a particular line. although it is definitely not good programming practice, it is possible to mix alpha and numeric values.

```
10 INPUT "NAME, BIRTHDAY"; A$, A
```

GET

A major problem with INPUT is that it does not allow real-time programming. All processing comes to a grinding halt while the user takes his time to enter some characters and strike RETURN. PET BASIC has been equipped with a special function which will yield one character at a time from the keyboard or tell if a key has been pressed.

The command is GET. GET is identical in syntax to INPUT. It is possible to specify a list of variables but

generally this is not a good idea because the purpose of GET is to scan the keyboard and return with a single key closure. When a numeric value is specified

```
GET A
```

only numeric keys will be accepted as input. All others will cause the message:

```
?SYNTAX ERROR
```

Use of the numeric value is confusing because if no key has been struck, the value returned is zero. Otherwise it will have a value 1-9 for keys 1-9.

The most desireable way to use GET is with a string variable. If a key has not been pressed, the string will have a null value (length = 0); otherwise the string will contain the character corresponding to the key that was pressed. See the next section for a detailed explanation of how strings work.

GET calls a routine which examines the keyboard interrupt buffer. If the buffer is empty, the variable contains a value of null or zero. If there are characters, the first is taken out of the queue and returned. Since the length of queue is 10 characters, calling GET 10 times in a loop is a good way to insure that the queue is empty when waiting for a response. This is particularly useful in interactive games.

The following routine will wait for a key to be pressed and exit only with the value of a key closure:

```
10 GET A$
20 IF A$ = ""THEN 10
```

In this case, ""is a literal which contains no characters and is a null string.

NOTES

We have been describing numeric functions primarily, but almost any useful program also has to deal with alphanumeric data. BASIC has a set of functions to deal with these data. Also, all alphanumeric data may be expressed as a continuous connection of characters which is viewed by BASIC as the value of a single variable.

PET BASIC, has a $ notation which is used to express variables which are strings of alphanumeric data. All of the rules which apply to normal variables apply to the string variable.

Following the naming conventions, we can create a variable A$ not equal to A% and not equal to A. Type A$ = "NOW IS THE TIME" and PRINT A$ to show the value of the string. This technique can define a string of a length up to about 70 characters, depending on the number of characters of the line number -- all that can be entered on a line. However, the limitation on the number of characters that can be stored in a string is 255. You can build strings larger than can be entered. The accumulation of characters from an I/O device and the construction of data is accomplished by the concatenation of strings. The operator that is used is + .

We can modify the expression A$ which we have been developing by typing A$ = A$ + " FOR ALL". Print A$ and you can see that the literal we typed in had a space at the beginning. Unlike numbers which are formatted by BASIC, the value of the literal is taken literally. A string can contain all combinations of bits including those that form control characters such as cursor down, and carriage return. This will be illustrated soon.

BASIC allows string expressions up to 255 characters long. These can be output to the screen or to any output device which accepts more than 79 characters. Input, however, is usually restricted to 79 characters because of the size of the input buffer. This problem can be handled by breaking strings into substrings before they are input or by using GET to input each character individually. The substrings or individual characters can then be recombined into the original string by concatenation.

COMPARISON OF STRINGS

The ASCII table is defined in Figure 2.6. It contains the order in which characters within the PET are represented when two strings are compared. Characters within a set of strings are compared starting at the leftmost character to the end of the field specified.

Using the ASCII table, we can compare a string containing an "A" to one containing a "B" in the same position. The result is that the second string is greater than the first.

A string containing a blank is less than a "1", which is less than an "A", which is less than a "B". The string "A" is less than the string "ABC" or any string containing "A" as the first character. All characters are compared in sequence with the first unequal character defining the relationship between the strings. Thus the same relational functions may be used for both strings and numbers.

< > for unequal
= for equal
< for less than
> for greater than

Immediately the string comparison feature can be applied to help you construct ordered lists such as a check file or a telephone directory. Comparisons can also be used to search ordered lists such as a file or a telephone directory.

Try the following program to develop a feeling for sequences and matching functions:

```
10 INPUT A$
20 INPUT B$
30 IF A$ = B$ THEN ? "A$ = B$" GOTO 10
40 IF A$ <$B THEN ? "A$ <B$" GOTO 10
50 PRINT "A$ >B$": GOTO 10
```

NUMBERS AND ASCII CODES

Two complementary pairs of operations on strings and numbers allow us to put unconventional things into character strings.

STR$

STR$ is a function of one argument. It returns a string that is the character representation of the numeric expression:

```
10X = 3.1
20 ?STR$ (X)
RUN
  3.1
READY
```

Positive numbers are preceded by a blank in the STR$ equivalent. Negative numbers have a sign in the corresponding position.

VAL

VAL is the complement of STR$. It converts a string to a number which may be used for computations. If the first
non-blank character of the string is not numeric, then the value of the function is zero.

```
?VAL("Z")
0
READY
```

On the other hand, VAL will convert as many digits as it can up to an invalid character.

```
?VAL ("3.14 AB")
3.14
```

VAL is an excellent function to use with INPUT since it can prevent an inexperienced user from causing a REDO from START.

CHR$

We have shown that strings may be assigned printable ASCII characters through either literals or direct INPUT, but some devices require control characters which cannot be produced by normal means. For example, a PET printer uses shifted carriage return as a specialterminator to indicate a carriage return with no line feed when it performs overprinting. CHR$ allows you to specify such control characters by giving the ASCII code number. CHR$ is a function to convert a number into internal ASCII representation. The value of the argument must be 0 = X< = 255.

```
10 A$ = CHR$(65) + CHR$(66)
20 PRINT A$
RUN
AB
READY.
```

In the above examples, 65 is the ASCII code for "A" and 66 is a "B". We converted the codes to characters before concatenating them and printing them out.

ASC

ASC turns a character into an ASCII code number which may be used in numerical calculations. The parameter is a string.

```
?ASC("A")
65
```

If the string consists of multiple characters, then this function will return the code for the first character of the string.

```
?ASC("123")
49
```

The ASCII code for "1" is 49.

SEGMENT OF STRINGS

In many cases it is desirable to access just part of a string in developing an ordered list. Consider the problem where in response to an INPUT, a person's name is typed in. It might consist of their first name, middle intitial, and last name. It is important that for sorting, however, that not all Johns be together, but that the list be ordered by last name.

In order to be able to separate parts of strings and use them in expressions, PET BASIC provides three functions. Most of your programming with strings will consist of using one of these three functions to analyze pieces of a constructed string. We will present the use of the functions and define all three at once as they are essentially the same function. Three combinations are provided mainly for programming convenience.

LEFT$, RIGHT$, and MID$

The function specified as LEFT$(string variable, I) gives the leftmost "I" characters of the string specified. If I
is negative, or zero, or greater than 255, then an ILLEGAL QUANTITY ERROR is printed. RIGHT$(STRING VARIABLE, I)gives the rightmost "I" characters of the string expression. When "I" is less than, or equal to zero,
or greater than 255, an ILLEGAL QUANTITY ERROR is printed.

There are two expressions for MID$. The first most general one is MID$(STRING VARIABLE, I, J). This expression gives "J" characters from the string starting with the "I"th character. If "I" is greater than the length of the string, then this will give a null string. If either "I" or "J" negative, or greater than 255, an ILLEGAL QUANTITY ERROR is printed. For "J" greater than the number of characters left in the string, all the characters from "I" to the end of the string are returned.

The second expression is MID$(STRING VARIABLE, I) which is the same as specifying a "J" greater than the length of the string. All the characters starting in the "I" position until the end of the string are returned. If "I" is greater than the length of the string, then a null string is returned and if "I" is negative, zero, or greater than 255, and ILLEGAL QUANTITY ERROR is printed.

All of these variables combined will define a new function which allows us to take either the left number of characters, right number of characters, or a given number of characters starting at a given position of the string.

To find the last name from our previous example, we can analyze characters starting from the rightmost character of the string until the first blank is encountered. To implement this program we need one more function.

LENGTH OF A STRING

The LEN function gives an exact count of the number of characters contained in a string. Non-printing

characters and blanks are all counted as part of length.

Strings are stored in BASIC with a 3-byte vector. Two bytes are a pointer to the location in memory where the string is stored and the third byte is the length, the LEN function extracts this byte.

We can now write a general purpose program to extract the last name from a full name.

```
10 INPUT"NAME:FIRST, MI, LAST";A$
20 I = LEN(A$)
30 IF mid$(A$,I,1) = " "THEN 60
40 I = I - 1
50 IF I>0 GOTO 30
60 PRINT "LAST NAME = "; MID$(A$, I + 1)
```

Two variants of MID$ are used here. Line 30 uses the case where a length is specified as the first parameter. We are using a length of 1 to search for the blank delimiting the last name. Line 60 does not specify a length in the MID$. Everything beyond the position of the blank is taken.

STRING STORAGE

Strings are stored in the space between the end of your BASIC program and the highest RAM locations. As each new string is added, a chain grows downward from the top of memory.

Storage is optimized by never creating a copy of a string assigned to a literal. In this case the vector for the string points to where the literal occurs in text in memory. Likewise, if an expression A$ = B$ is executed, both A$ and B$ will share the same copy of the string. New string is required only if a concatenation or INPUT is executed.

A LARGER EXAMPLE OF STRING FUNCTIONS

Using the string functions described thus far we can write a routine which will shuffle a deck of cards for us and deal them out one at a time. The following routine has applications in many games like poker or bridge. Note use of the PET graphics card symbols:

```
109 REM SET UP DECK WITH ALL 52 CARDS
110 C$="A♠2♠3♠4♠5♠6♠7♠8♠9♠T♠J♠Q♠K♠"
120 C$=C$+"A♥2♥3♥4♥5♥6♥7♥8♥9♥T♥J♥Q♥K♥"
130 C$=C$+"A♦2♦3♦4♦5♦6♦7♦8♦9♦T♦J♦Q♦K♦"
140 C$=C$+"A♣2♣3♣4♣5♣6♣7♣8♣9♣T♣J♣Q♣K♣"
190 REM PULL A CARD
200 R=2*INT(LEN(C$)*RND(1)/2+1)-1
201 N$=MID$(C$,R,1):Y$=MID$(C$,R+1,1)
430 REM SHRINK THE DECK
432 IFR>1THENT$=LEFT$(C$,R-1):GOTO435
433 T$=""
435 C$=T$+MID$(C$,R+2)
439 REM PRINT A CARD
440 PRINTN$;Y$,
450 IFLEN(C$)>=1THEN200
455 REM END OF DECK
460 INPUT"ANOTHER DEAL    ";Z$
470 GOTO105
READY.
```

The string C$ is initialized to contain a deck of cards. Two characters represent each card; the suit and rank. As a card is dealt, N$ contains the rank and Y$ contains the suit. The deck string, C$, shrinks each time so that unique cards are always dealt.

Statement 105 clears the screen. This is done just for show so that the program can illustrate the dealing of cards. C$ is initialized in statements 110 through 140. C$ is concatenated because the literal assignment is too large to fit on one line.

Statement 200 uses RND to generate an index into C$. The random index is in the range 1 to LEN(C$) – 1. In 201 the index is used to pull N$(rank)Y$(suit) from C$ by the MID$ function.

432 through 435 removes the card from the string so that it will not be dealt again. Since the second argument of LEFT$ cannot be zero, the R>1 test in 432 prevents an ILLEGAL QUANTITY ERROR.

440 prints each card for our program as it is pulled. 450 tests for the end of the deck and 460 allows the user to reshuffle.

USER DEFINABLE FUNCTIONS

To this point we have covered all the functions intrinsic to BASIC. Those familiar with mathematics are used to many more functions in that realm, especially trigonometric. While one could write code to approximate certain functions in line it becomes very tedious and from a documentation standpoint a simple expression might become unreadable. Fortunately ,the facility exists in PET BASIC to define functions in terms of other functions.

A function is defined in a DEF statement:

```
100 INPUT B
110 INPUT C
120 DEF FN A(V) = V/B + C
```

The name of the function is "FN" followed by any legal variable name. Recall that a variable is either a letter or a letter followed by a letter or digit.
Thus the following are valid function names:

```
FNX
FNJ7
FNKO
FNR2
```

The most severe limitation of user-defined functions is that they must be contained in their entirely on one line (80-characters). String functions cannot be defined.

The variable in parentheses following the variable name is called a dummy variable. A function may be defined to be any expression but it may have only one argument. Other variables used in the expression are considered to be global (have the same value as in the rest of the program), and their current values are used in the evaluation.

After the funtion definition has been executed, a user defined function can be used as in the following example:

```
130 Z = FNA(3)
140?Z
```

When the DEFFN statement is executed, a simple variable entry is made in the variable table. The first character of the name has bit 7, the most significant bit, set to indicate it is a function name. Associated with the name are two pointers: an address of the text where the function is stored and an address of where the dummy variable is stored. The code to execute a function is re-entrant so that a function may be defined in terms of other DEF FN. An out of memory error will occur in time as the available stack space is consumed by recursion.

Figure 6.1 shows some user-defined functions which are ready to be used in PET BASIC programs.

FUNCTIONS EXPRESSED IN TERMS OF BUILT-IN BASIC FUNCTIONS

SECANT, SEC(X)

```
DEF FNA(X) = 1/COS(X)
    FOR X < >π/2
```

COSECANT, CSC(X)

```
DEF FNB(X) = 1/SIN(X)
    FOR X< >0
```

CONTANGENT, COT(X)

```
DEF FNC(X) = COS(X)/SIN(X)
    FOR X< >0
```

INVERSE SINE, ARCSIN(X)

```
FND(X) = ATN(X/SQR(-X*X+1))
    FOR ABS(X) < 1
```

INVERSE COSINE, ARCCOS(X)

```
DEF FNE(X) = -ATN(X/SQR(-X*X+1))+π/2
    FOR ABS(X) < 1
```

INVERSE SECANT, ARCSEC(X)

```
DEF FNF (X) = ATN(SQR(X*X-1)) + (SGN(X)-1)*π/2
    FOR ABS(X) > 1
```

INVERSE COSECANT, ARCCSC(X)

```
DEF FNG(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*π/2
    FOR ABS(X) > 1
```

INVERSE COTANGENT, ARCCOT(X)

```
DEF FNH(X) = -ATN(X)+π/2
    FOR ANY X
```

HYPERBOLIC SINE, SINH(X)

```
DEF FNI(X) = (EXP(X)-EXP(-X))/2
    FOR ANY X
```

HTPERBOLIC COSINE, COSH(X)

```
DEF FNJ(X) = (EXP(X)+EXP(-X))/2
    FOR ANY X
```

HYPERBOLIC TANGENT, TANH(X)

```
DEF FNH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1
    FOR ANY X
```

HYPERBOLIC SECANT, SECH(X)

```
DEF FNL(X) = 2/(EXP(X)+EXP(-X))
    FOR ANY X
```

HYPERBOLIC COSECANT, COSH(X)

```
DEF FNM(X) = 2/EXP(X) - EXP(-X))
    FOR X < > 0
```

HYPERBOLIC COTANGENT, COTH(X)

```
DEF FNN(X) = EXP(-X)/(EXP(X)+EXP(-X))*2+1
    FOR X < > 0
```

INVERSE HYPERBOLIC SINE, ARCSINH(X)

```
DEF FNO(X) = LOG(X + SQR(X*X + 1))
    FOR ANY X
```

INVERSE HYPERBOLIC COSINE, ARCCOSH(X)

```
DEF FNP(X) = LOG(X + SQR(X*X - 1))
    FOR X > =1
```

INVERSE HYPERBOLIC TANGENT, ARCTANH(X)

```
DEF FNQ(X) = LOG((1 + X)/(1 - X))/2
    FOR ABS(X) < 1
```

INVERSE HTPERBOLIC SECANT, ARCSECH(X)

```
DEF FNR(X) = LOG((SQR(-X*X + 1) + 1)/X)
    FOR 0 < X < =1
```

INVERSE HYPERBOLIC COSECANT, ARCCOSH(X)

```
DEF FNS(X) = LOG((SGN(X)*SQR(X*X + 1) + 1)/X)
    FOR X < > 0
```

INVERSE HYPERBOLIC COTANGENT, ARCCOTH(X)

```
DEF FNT(X) = LOG((X + 1)/(X - 1))/2
    FOR ABS(X) > 1
```

GOSUB-RETURN

We have seen how to use the DEF FN to create a single variable function which can be used like any intrinsic function. The major limitation of DEF FN is that it can consist of only a single algebraic expression and it must fit on one line.

Often several lines of code will be repeated through a program. These program lines can be collected in one place and executed by a GOSUB command:

```
GOSUB 5000
```

The lines of code are called a subroutine. GOSUB means go to the subroutine. It differs from GOTO in that GOSUB remembers at which line number it was executing before the GOSUB and can return automatically to the following line after executing the subroutine code.

A subroutine is stored as a series of lines in BASIC starting at the line number specified by the GOSUB. The last line of the subroutine must be a RETURN statement. This tells BASIC you want to resume executing the mainline code after the GOSUB.

Example;

```
10 REM THIS IS THE MAINLINE CODE
20 GOSUB 50
30 STOP
50 REM THIS IS A SUBROUTINE
60 RETURN
```

If we could take a snapshot of execution, we would see the lines executed in this order

10 - 20 - 50 - 60 - 30

Five bytes are pushed onto the stack when a GOSUB is executed: a GOSUB token, and two bytes each for the line number and text address of the GOSUB. The line number following the GOSUB is stuffed into the currently executing line number and the GOTO routine handles the branch. RETURN restores the line number and text address from the stack to resume mainline execution. All F O R entries in front of the GOSUB entry are also eliminated.

The physical limitation on the number of GOSUB's in effect at one time is 23. After this many there is very little stack space left.

Example of subroutines

Consider the factorial function:

n! = 1 x 2 x 3 x...xn

You cannot define this function with the DEF FN command. On the other hand, you can use the following simple routine to find n! for any given n (up to 34). **(NF denotes n factorial)**

```
10      INPUT N
100     I = 1:NF = 1
110     NF = NF*I
120     I = I + 1
130     IF I <= N GOTO 110
140     PRINT NF
```

The routine on lines 100-140 could be used many times during a program using different values for N. For example, suppose you want a binomial coefficient:

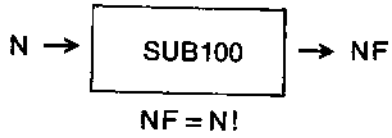
$$\binom{m}{r} \quad \frac{m!}{r!\,(m-r)!}$$

The program would be

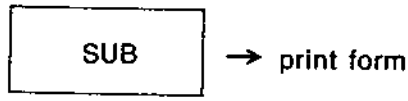
```
10      PRINT "M = ";: INPUT M
15      PRINT "R = ";: INPUT R
20      N = M:GOSUB100:X = NF
30      N = R:GOSUB100:Y = NF
40      N = M - R:GOSUB100:Z = NF
50      BC = X / ( Y*Z )
60      PRINT BC
70      END
100     I = 1:NF = 1
110     NF = NF*I
120     I = I + 1
130     IF I <= N GOTO 110
140     RETURN
```

TYPE RUN
for the values M = 11 R = 6.
RESULT IS 462

Subroutines act like a "black box" or complex function within the program. Certain fixed variables are used to input the data and other fixed variables (or sometimes the same variable) are used to output the results. For example, in the subroutine on lines 100-140, the variable N is input and the variable NF is output as shown:



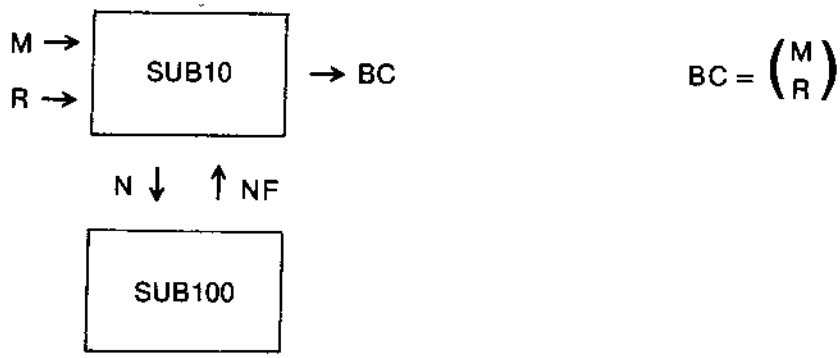
When we make N equal to M, R, and M-R respectively, we get NF equal to M!, R! and (M-R)!.
Of course, some subroutines do not need inputted variables as they might just perform a specified function such as printing a special form on the screen:



NESTED SUBROUTINES
The subroutine on page 6-14 itself could be used as a subroutine in a program that repeatedly calculates the binomial coefficient. Merely change line 70 to

70 RETURN

The subroutine, denoted SUB 10, beginning on line 10 and ending on line 70 has the following structure:



Subroutines that are used by other subroutines are called nested subroutines. In this case, SUB100 is nested in SUB20. Many programs have subroutines nested in subroutines in nested subroutines...The only limit is the amount of memory available.

Subroutines can also be nested in more than one subroutine. An input subroutine, for example, that accepts specific characters from the keyboard, prints a winking cursor, and prints the given characters on the screen, might be called on many times in the main program itself and also in various other subroutines.

CAUTIONS
A common error in using subroutines is to allow a mainline execution to fall into a following subroutine and result in a RETURN WITHOUT GOSUB ERROR. Put a STOP or END statement in your code to prevent

this

```
10 GOSUB 20
20 RETURN
```

```
10 GOSUB 20
15 END
20 RETURN
```

Sometimes, you might have a tendency to make everything into a subroutine. If a given subroutine is used just once, then it should be incorporated into a program where it is used to save execution time and memory space. On the other hand, subroutines are incredibly powerful programming tools and allow you to structure your program into blocks.

FOR-NEXT LOOPS

FOR-NEXT simplifies the writing of BASIC programs by allowing one to specify complex loop structures with a single statement.

FOR I = A TO B STEP C

The end of the loop is specified by the statement

NEXT

Nested FOR NEXT loops are permitted as long as each loop uses a unique variable. Use of identical loop variable names may result in NEXT WITHOUT FOR errors.

Exiting a FOR-NEXT loop via a branch will leave the FOR entry on the stack. The best way to handle this is to assign the maximum limit to the variable then exit the loop through a NEXT.

We have seen how repeated operations can be performed using a counting variable such as I in the routine.

```
10  I=1
20  I=I+1
30  IF I <=10 THEN GOTO 20
```

In this case, any routine appearing in lines 21-29 will be repeated 10 times. In addjtion, the variable I will have values which range from 1 to 10 in increments of 1.

This looping process can be genralized in the case:

```
10  I=A
20  I=I+C
30  IF I <=B THEN GOTO 20
```

The values of I will range from A to B in increments of size C.

Since this process is cumbersome to use, BASIC also provides you with the FOR-NEXT statement:

```
10  FOR I=A TO B STEP C
20  NEXT
```

I is the counting variable, A is the initial value, B is the ending value, and C is the increment.

A, B, C may not only be constants, but they can be any valid arithmetic expression

```
10 FOR I=A(2)+1 TO J*2 STEP -1
```

On the other hand, the counting variable can be any floating variable but cannot be integer (I%) or subscripted I(1,4). When the increments are of size 1(C = 1) you need not include the STEP in the program.

```
10  REM COMPUTATION OF FACTORIAL
20  NF=1
30  FOR I=1 TO N
```

```
40  NF=NF*I
50  NEXT
```

Note how much shorter and more clearly this routine is written compared to the same factorial computing program written without FOR-NEXT.

Whenever a FOR is executed, a 16-byte entry is pushed onto the stack. Before this is done, a check is made to see if there are any entries already on the stack for the same loop variable. If so, that FOR entry and all other FOR entries that were made after it are eliminated from the stack. This is done so that a program which jumps out of the middle of a FOR loop again will not use up 16-bytes of stack space each time.

NEXT matches the most recent stack entry or the variable specified as a parameter and resets the stack to that point. If no match is found, a NEXT WITHOUT FOR error occurs.

GOSUB execution also puts a 5-byte entry on the stack. When RETURN is executed, the stack is searched for a FOR entry that cannot be matched, When all the FOR entries on the stack have been searched, a pointer
is left on a GOSUB entry. This assures that if you GOSUB to a section of code in which a FOR loop is entered but never existed, the RETURN will still be able to find the most recent GOSUB entry.

RETURN eliminates the GOSUB stack entry and all FOR entries made after the GOSUB entry.

NESTED FOR-NEXT LOOPS
FOR-NEXT loops, like subroutines, can be nested, That is, a FOR-NEXT loop may be contained in another and so on. When doing so, it is important not to use the same counting variable as this will result in

```
?NEXT WITHOUT FOR ERROR
10 FOR I=1 TO 10
15 PRINT "I"
20 FOR J=1 TO 10
25 PRINT "J"
30 FOR K=1 TO 10
35 PRINT "K"
40 NEXT
50 NEXT
60 NEXT
```

Lines 40-60 of the above example are confusing at first glance because one cannot tell which NEXT corresponds to which FOR. Optionally one may specify a variable following NEXT. The variable refers to the counting variable used in the corresponding FOR but in no way is it required by BASIC to execute the NEXT.

```
40 NEXT K
50 NEXT J
60 NEXT I
```

PET BASIC will also allow you to write one NEXT that terminates all three FORs at one time

```
40 NEXT K, J, I
```

A NEXT WITHOUT FOR error will result, however, if you are careless in specifying the order of K,J,I.

It is interesting, however, to see how compact the notation appears and how powerful the FOR-NEXT expressions can be when they are nested.

Some hints

You may change the value of the counting variable during the looping sequence.

For example,

```
10 FOR I = 1 TO 8
20 X = X + 1
30 IF I = 7 THEN I = 8
40 NEXT
50 PRINT X
```

will compute the value

$$X = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$$

Similarly, when you exit a FOR-NEXT loop using a branch, you should assign the counting variable the end value and then exit the loop via a NEXT statement. For instance, you should use

```
10 FOR I = 1 TO 10
20 IF FNA(I) = 0 THEN I = 10
30 NEXT:RETURN
```

instead of

```
10 FOR I = 1 TO 10
20 IF FNA(I) = 0 THEN RETURN
30 NEXT
```

SUBSCRIPTED VARIABLES

Array variables need not be declared with a DIM statement if they have only one dimension and contain less than 10 elements. The total number of elements in an array can be computed by multiplying the (number of elements in each dimension) + 1 by the other subscripts. Thus A(9,8) contains (9+1)*(8+1) elements. Subscripts start at 0 and go up to the maximum value

A(0,0)-------A(0,8)

A(9,0) A(9,8)

Limits on the number of dimensions and size of a dimension are determined by size of memory available and space available on a line following a DIM. PET BASIC restrict the total number of array elements to 256. Each array element requires at least 5-bytes of storage.

If a single dimension array requires more than 10 elements, the DIM statement must be executed before the first reference. Otherwise, a REDIM'ED ARRAY error will occur.

Example: List of account balances

1	$100
2	$135
3	$57.86
4	<$987>
5	$22
6	<$63>
7	$50
8	<$21>
9	$21

Suppose we need to write a simple program which allowed you to INPUT an account number and a transaction and keep a running total on each account. We could refer to each account balance as A1, A2,

A3, A4, A5, etc. This is acceptable but would require a lot of parallel logic to accomplish the summation

```
10 INPUT "ACCOUNT, CHARGE"; I, C
20 IF I = 1 THEN A1 = A1 + C
30 IF I = 2 THEN A2 = A2 + C
etc.
```

This list can be stored in a single variable which is actually a list of variables. This list is an array of values and an individual value is accessed by an index. The index we can use is the account number. Our program can be reduced to:

```
10 INPUT "account, charge"; I,C
20 A(I) = A(I) + C
30 GOTO 10
```

The list we have represented has 9 rows and 1 column. Thus it is a 1 dimensional array. A multiple column table can also be represented. This is a two dimensional array.

Account #	Balance	# of transactions
1	$100	1
2	$135	1
3	$57.86*	1
4	<$987>	1
5	$22	1
6	<$63>	1
7	$50	1
8	<$21>	1
9	$21	1

Our table has 9 rows and 2 columns. To access a certain entry position, you must specify the row index and column index of where it is contained. For example, the quantity denoted by a * is in row 3, column 1.

In order to use such a table in a BASIC program, you must provide a statement, to describe the number of rows and columns contained in the array variable.

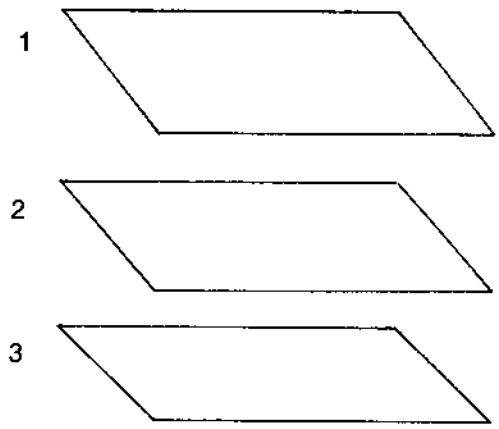
Such a description is a DIMension statement. For our table of 9 rows and 2 columns we could write

```
DIM A(9,2)
```

Let us rewrite our program to update the column containing the number of transactions

```
10 INPUT "ACCOUNT, CHARGE"; I, C
20 A(I,1) = A(I,1) + C
30 A(I,2) = A(I,2) + 1
40 GOTO 10
```

Now suppose that we had a table for each of 5 companies and each company had 9 accounts and each account had a balance and each balance had a number of transactions. We can describe this as piling sheets of paper on top of each other and refering to each sheet by number.



We have created by this example a multi-dimensional subscripted variable. These arrays correspond to matrices used in mathematics.

In mathematics, a vector is an ordered collection of numbers:

$v=(v_1, v_2,...,v_n)$

The above vector has n components and is called a vector of dimension n.

For example,

$v=(3,9,2)$

is a vector of dimension 3.

Order is imporatant here since if

w=(3,2,9)

$w \neq v$.

Vectors can be stored in memory using subscripted variables. These variables are used in the same way as the variables we have seen so far -X, I%, A$, etc. That is, they call whatever value is stored in that variable or return a zero or null (" ") if the value has not been previously specified.

Like vectors, subscripted variables have the power to execute a large number of operations using a single notation. They are especially useful when combined with FOR-NEXT loops as the next example shows.

Example: Dot Product

The dot product of two vectors v & w is a vector, denoted by v • w, whose ith component $(v \bullet w)_i$ is $v_i \times w_i$.

For example, in the four dimensional case, if

$v = (v_1, v_2, v_3, v_4)$

and $w = (w_1, w_2, w_3, w_4)$

Then $v \bullet w = (v_1 \times w_1, v_2 \times w_2, v_3 \times w_3, v_4 \times w_4)$

Suppose we had

$v = (5,6,7,11,4,6,1)$

$w = (9,5,2,1,0,3,2)$

Then a program to compute the dot product v • w might look like

```
FOR I = 1 TO 7:READ V(I):NEXT
FOR I = 1 TO 7:READ W(I):NEXT
FOR I = 1 TO 7:VW(I) = V(I)*W(I):NEXT
FOR I = 1 TO 7:?VW(I):NEXT
```

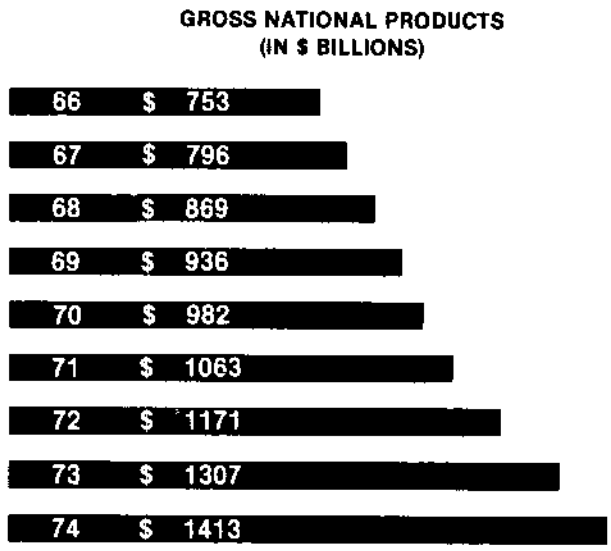
DATA 5,6,7,11,4,6,1
DATA 9,5,2,1,0,3,2

SUBSCRIPTED STRING VARIABLES

It was mentioned previously that subscripted variables can be

decimal: A(I)
integer: A%(I)
string: A$(I)

Subscripted string variables are extremely useful as shown in the next program which prints a bar graph of the U.S. GNP from 1966 through 1974.



The program listing is:

```
READY
  10 SPACE $ = "                                "
  20 FOR I = 1 TO 7: READ A$(I):NEXT
  30 FOR I = 0 TO 8: READ V(I):NEXT
  40      PRINT" ▽ SPC(8)"GROSS NATIONAL PRODUCTQ"

  50 PRINTSPC(12)"(IN $BILLIONS)Q"
  100 FOR I = 0 TO B
  110 X = V(1)/45:Y = INT(X)
  120 PRINT "R"LEFT$(SPACE$,Y)A$(S*(X – Y))
  130Q" PRINT " ▭ R"STR$(66 + I)" $"STR$(V(I))"Q"

  140 NEXT
  200 DATA"■","■","■","■","▌","|","]"
  210 DATA753,796,869,936,982,1063,1171,1307,1413
  READY
```

The subscripted values V(0), V(1),...,V(8) are the GNP's for each of the 9 years. The subscripted strings A$(0), A$(1),...A$(7) give accuracy to the graph by printing these graphics:

string	prints	ASC
A$(0)	null(by default)	
A$(1)	▌	165
A$(2)	■	180
A$(3)	■	181
A$(4)	■	161
A$(5)	■	182 (R)
A$(6)	■	170 (R)
A$(7)	■	167 (R)

THE HEADING

GROSS NATIONAL PRODUCT
(IN $BILLIONS)

is printed in lines 50 and 60 and then a FOR-NEXT loop on lines 100-140 prints out the eight bars. Line 120 prints out each bar and line 130 prints a cursor up and then the associated year, STR$(66 + I) and GNP, STR$(V(I)).

Each bar is made up of Y reverse field spaces and the string A$(8*(X-Y)). The Y is determined by the formula

Y = INT(V(I)/45)
= INT (GNP/45)

Here, 45 is purely a scale adjustment. The proportions of the bars remain the same when values other than 45 are used.

Fine tuning on the bar length is accomplished using the subscripted string variable

A$(8*(X-Y))

Here 8*(X-Y) will range over the decimal values 0 through 7.99...9 but A$ automatically truncates the decimal part.

DIMENSION STATEMENTS

When using more than 10 subscripts for any variable, a dimension statement must be given. It takes the form, DIM A$(K), where K is the largest subscript of A$ used in the program. When variables are redimensioned without a CLR statement or when a dimension statement appears after the variable has been used, a ?REDIM'D ARRAY ERROR occurs. When a dimension statement is made, space is reserved in memory for the given number of variables, including the variable whose subscript is 0. It is good programming sense, therefore, to begin subscripts at 0 and not 1.

Because the variables are divide in storage between arrays and simple variables insertion of an additional simple variable is a bit more complicated once an array has been defined. First, the entire array storage area must be block moved upward by seven bytes and the pointers adjusted upward +7. Finally, the simple variable can be inserted at the end of simple variable storage.

If large arrays are defined and initialized first before simple variables are assigned, much execution time can be lost moving the arrays each time a simple variable is defined. The best strategy to follow in this case is to assign a value to all known simple variables before assigning arrays. This will optimize execution speed.

Functions of new and CLR on data pointer:

CLR

String pointer equated to top of memory
Data pointer to start of text -1
End of array table to start of variables
End of simple variables to start of variables

NEW

String pointer equated to top of memory
Data pointer to start of text -1
End of array table to start of text +3
End of simple variables to start of text +3
Start of variables to start of text +3

PRINCIPAL POINTERS INTO PET RAM

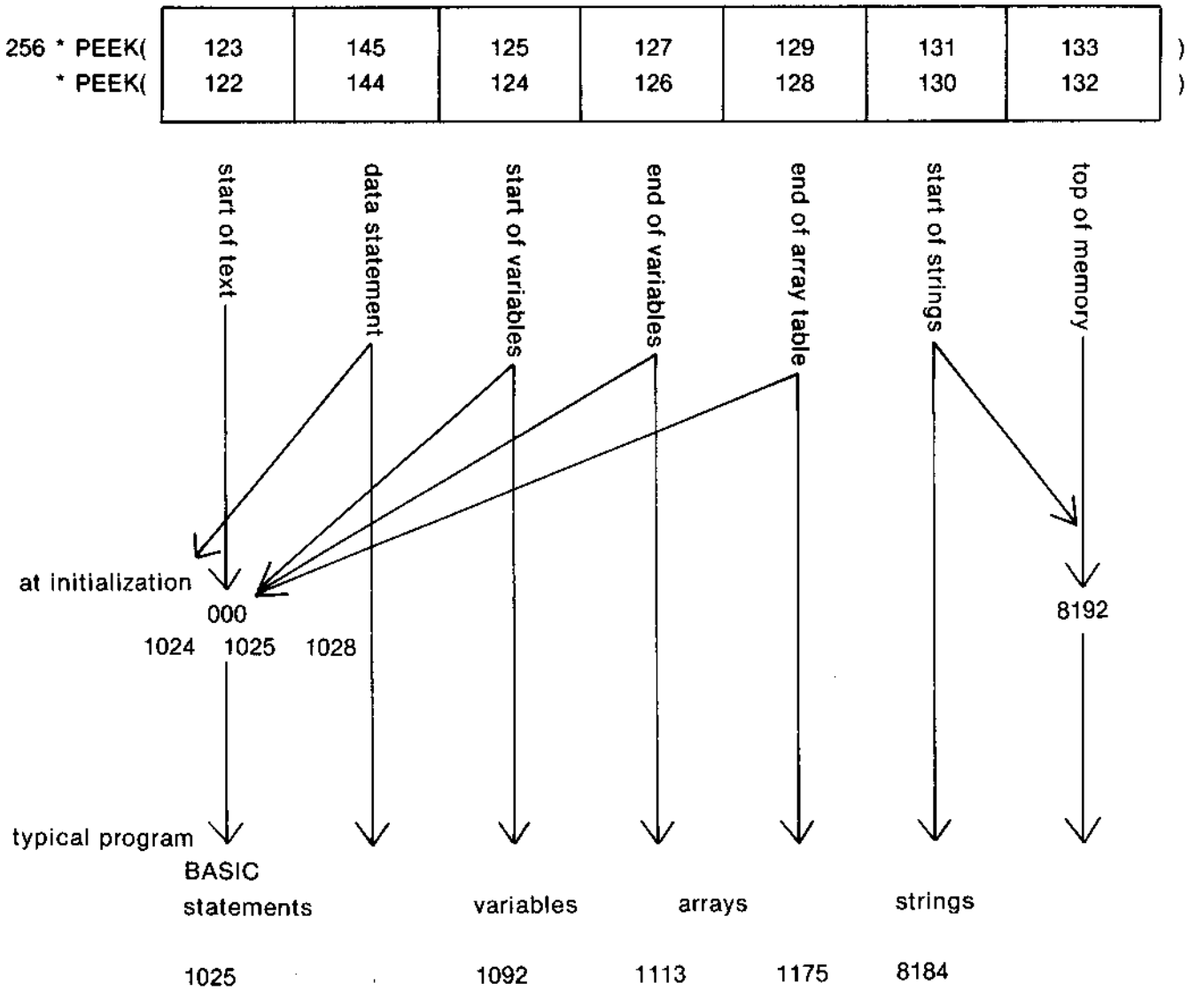


Figure 6.2. Principal pointers into PET RAM

As indicated in Figure 7.1, there are four connectors provided, accessible through slots in the rear and side of the PET that enable the user to interface the computer with external devices.

As outlined in Figure 7.2, edge card connectors are utilized which are, in fact, direct extensions of the PET main logic assembly board itself. There are two contacts to each position of the connector. The contact identification convention for J1 and J2 is also illustrated in Figure 7.2.

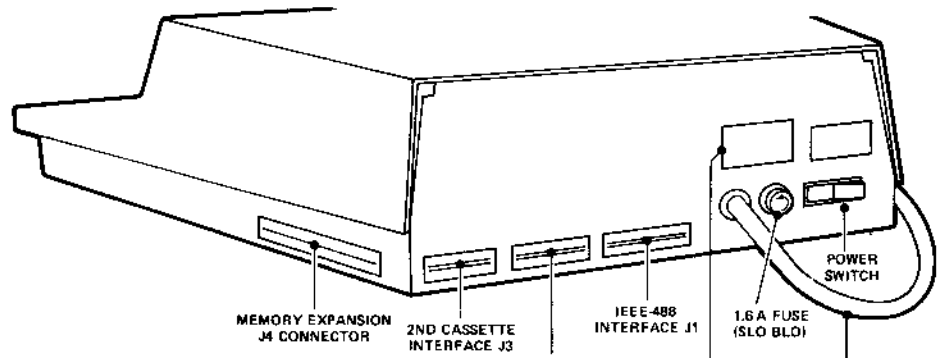


Figure 7.1. Simplified view of PET showing switch, fuse, line cord and interfacing connectors.

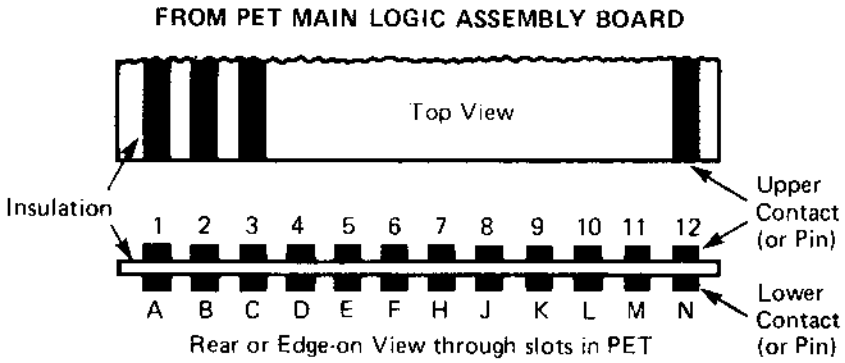


Figure 7.2. Simplified views of edge connectors J1 and J2 to illustrate contact identification convention.

IEEE-488 INTERFACES (Connector J1)

The standard IEEE-488 connector is not used on the PET. Instead, a standard 12 position, 24 contact edge connector with .156 inch spacing between contact centers is provided. This permits it to be compatible with all of the other connections to the PET.

Keying slots are located between pins 2-3 and 9-10.

Table 7.3 shows the PET contact identification characters, the connection for a standard IEEE connector,

the IEEE mnemonics and the signal definitions.

Electrical drive capability and line impedance matching is in accordance with IEEE-488 specifications.

PET Pin Characters	Standard IEEE Connector Pin Numbers	IEEE Signal Mnemonic	Signal Definition/Label
Upper Pins			
1	1	DI01	Data input/output line #1
2	2	DI02	Data input/output line #2
3	3	DI03	Data input/output line #3
4	4	DI04	Data input/output line #4
5	5	EOI	End or identify
6	6	DAV	Data valid
7	7	NRFD	Not ready for data
8	8	NDAC	Data not accepted
9	9	IFC	Interface clear
10	10	SRQ	Service request
11	11	ATN	Attention
12	12	GND	Chassis ground and IEEE cable shield drain wire
Lower Pins			
A	13	DI05	Data input/output line #5
B	14	DI06	Data input/output line #6
C	15	DI07	Data input/output line #7
D	16	DI08	Data input/output line #8
E	17	REN	Remote enable
F	18	GND	DAV ground
Lower Pins			
H	19	GND	NRFD ground
J	20	GND	NDAC ground
K	21	GND	IFC ground
L	22	GND	SRQ ground
M	23	GND	ATN ground
N	24	GND	Data ground (DI01-8)

Table 7.3. PET contact identification characters. IEEE-488 identification characters, associated labels and descriptions.

RECEPTACLES FOR THE IEEE INTERFACE

A list of frequently used 12 position, 24 contact receptacles that are suitable for connection to the PET edge card connector J1 and J2 is shown here:

Manufacturer	Part Number
Cinch	251-12-90-160
Sylvania	6AG01-12-1A1-01
Amp	530657-3
Amp	530658-3
Amp	530654-3

Table 7.4. Receptacles recommended for PET IEEE-488 connectors or parallel user port.

IEEE-488 CONNECTORS

The IEEE-488 standard receptacles are *not* directly connectable to the PET edge connector; some of these are shown in Table 7.5, and belong to the Cinch Series 57 or Champ Series (Amphenol). Also shown are their matching plugs.

Connector Manufacturer	Identifier	Description
Cinch	571Ø24Ø	Solder-plug
Cinch	572Ø24Ø	Solder-receptacle
Amp	552Ø1-1	Insulation displacement plug
Amp	552Ø5-1	Insulation displacement receptacle

Table 7.5. IEEE standard connectors

Commodore has available a 1 meter long IEEE-488 dual connector-PET edge connector, cable. Please contact your local dealer or Commodore for price and delivery.

PARALLEL USER PORT (Connector J2)

The lines for this interface are brought out from the PET main logic board to a 12 position, 24 contact edge connector with a .156 inch spacing between contact centers. See Table 7.4 for suitable mating connectors.

Keying slots are located between pins 1-2 and 10-11.

Table 3-1 shows the PET pin identification characters, the corresponding labels and their descriptions.

Note that the connections 1-12, the top line of contacts (see Figure 7.6), are primarily intended for use by the PET service department or qualified dealers. When using the incorporated ROM diagnostic, a special connector is used; this jumpers some of the top contacts to the bottom contacts. *It is strongly advised that the top connectors 1-12 be used only with extreme caution.*

Pin Identification Character	Signal Label	Signal Description
1	Ground	Digital ground.
2	T.V. Video	Video output used for external display, used in diagnostic routine for verifying the video circuit to the display board.
3	IEEE-SRQ	Direct connection to the SRQ signal on the IEEE-488 port. It is used in verifying operation of the SRQ in the diagnostic routine.
4	IEEE-EOI	Direct connection to the EOI signal on the IEEE-488 port. It is used in verifying operation of the EOI in the diagnostic routine.
5	Diagnostic Sense	When this pin is held low during power up the PET software jumps to the diagnostic routine, rather than the BASIC routine.

Table 7.6. Parallel user port information. PET pin identification characters, the corresponding signal labels and their descriptions.

Table continued on next page.

Table 7.6. Parallel user port information (continued).

Pin Identification Character	Signal Label	Signal Description
6	Tape #1 READ	Used with the diagnostic routine to verify cassette tape #1 read function.
7	Tape #2 READ	Used with the diagnostic routine to verify cassette tape #2 read function.
8	Tape Write	Used with the diagnostic routine to verify operation of the WRITE function of both cassette ports.
9	T.V. Vertical	T.V. vertical sync signal verified in diagnostic. May be used for external TV display.
10	T.V. Horizontal	T.V. horizontal signal verified in diagnostic may be used for TV display.
11, 12	GND	Digital ground.
A	GND	Digital ground.
B	CA1	Standard edge sensitive input of 6522VIA.
C	PA0	Input/output lines to peripherals, and can be programmed independently of each other for input or output.
D	PA1	
E	PA2	
F	PA3	
H	PA4	
J	PA5	
K	PA6	
L	PA7	
M	CB2	Special I/O pin of VIA.
N	GND	Digital ground.

VERSATILE INTERFACE ADAPTER

The lines on the bottom side of the user port connector originate from a Versatile Interface Adapter (VIA MOS Technology part #6522).

The signals CA1, PA0-7, and CB2, are directly connected to a standard 6522 VIA located at hexadecimal address E840. (Decimal address 59456).

The parallel port consists of eight programmable bi-directional I/O lines PA0-7, an input handshake line for the eight lines,CA1, which can also be used for other edge-sensative inputs and a very powerful connection, CB2. This has most of the abilities of CA1, but can also act as the input or output of the VIA shift register.

A detailed specification for the VIA is below. All signals on the VIA that are not connected to the user port are utilized by the PET for internal controls. Please note that the user should avoid interfacing these signals in any way.

Table 7.7 shows the decimal and hexadecimal addresses in the PET associated with the VIA.

Decimal	Hexa-Decimal	$E840+	Addressed Location
59456	E840	0000	Output register for I/O port B.
59457	E841	0001	Output register for I/O port A with handshaking.
59458	E842	0010	I/O Port B Data Direction register.
59459	E843	0011	I/O Port A Data Direction register.
59460	E844	0100	Read Timer 1 Counter low order byte Write to Timer 1 Latch low order byte.
59461	E845	0101	Read Timer 1 Counter high order byte. Write to Timer 1 Latch high order byte and initiate count.
59462	E846	0110	Access Timer 1 Latch low order byte.
59463	E847	0111	Access Timer 1 Latch high order byte.
59464	E848	1000	Read low order byte of Timer 2 and reset Counter interrupt. Write to low order byte of Timer 2 but do not reset interrupt.
59465	E849	1001	Access high order byte of Timer 2; reset Counter interrupt on write.
59466	E84A	1010	Serial I/O Shift register.
59467	E84B	1011	Auxiliary Control register.
59468	E84C	1100	Peripheral Control register.
59469	E84D	1101	Interrupt Flag register (IFR).
59470	E84E	1110	Interrupt Enable register.
59471	E84F	1111	Output register for I/O Port A, without handshaking.

Table 7.7. VIA 6522 Decimal and Hexadecimal addresses in PET.

PROGRAMMING THE USER PORT

Data lines PA0-7 are individually programmed to function for input or output as required. This is done by using a software POKE 59459 command to place a number into the data direction register. Table 7.8 shows a practical example of input/output selection.

The programming need only be carried out at the beginning. From then on POKE 59471 can be used to drive the pins programmed as outputs, and PEEK(59471) will read all the inputs.

Command Statement	Binary Representation	Lines	Mode
POKE 59459,255	11111111	PA0-7	Output
POKE 59459,0	00000000	PA0-7	Input
POKE 59459,240	11110000	PA0-3	Input
		PA4-7	Output

Table 7.8. Parallel user port example.
Programming of lines PA0-7 for input/output operation.

SECOND CASSETTE INTERFACE (Connector J3)

This interface is brought out from the PET main logic board to a 6 position, 12 contact edge connector with .156 inch spacing between contact centers (See Figure 7.9).

A keying slot is located between pins 2-3.

This port is intended for use with the Commodore second cassette system only. Any other connections are made at the risk of the user. Please note that +5 volts is *not* intended for use as an external power supply.

Table 7.10 shows the PET pin identification characters, labels and descriptions. Table 7.11 shows some typical receptacles that are suitable for the second cassette connector.

FROM PET MAIN LOGIC ASSEMBLY BOARD

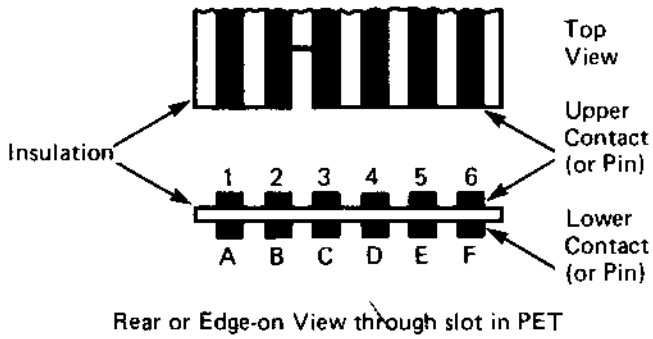


Figure 7.9. Simplified view of edge connector J3 with contact identification.

Note A-1, B-2, etc., imply a pin A to pin1, pin B to pin 2, connection.
In some special units, pins 1 through 6 were not connected.

Pin Identification Characters	Label	Description
A-1	GND	Digital ground.
B-2	+5	Positive 5 volts to operate cassette circuitry only.
C-3	Motor	Computer controlled positive 6 volts for cassette motor.
D-4	Read	Read line from cassette.
E-5	Write	Write line to cassette.
F-6	Sense	Monitors closure of mechanical switch on cassette when any button is pressed.

Table 7.10. Second cassette interface port.
PET pin identification characters, labels and associated descriptions.

Manufacturer	Identifier
Sylvania	6AJØ7-6-1A1-Ø1
Viking	2KH6/1AB5
Viking	2KH6/9AB5
Viking	2KH6/21AB5
Amp	53Ø692-1
Sullins	ESM6-SREH
Cinch	25Ø-Ø6-9Ø-17Ø

Table 7.11. A selection of suitable receptacles for connecting with the PET second cassette edge connector J3.

MEMORY EXPANSION CONNECTOR (Connector J4)

The memory expansion connector provides access to the buffered and decoded input/output lines from the 6502 microprocessor. Figure 7.12 shows a simplified view of the 40-position-80 contact edge connector used. The spacing between contact centers is 0.1 inch.

Note that the 40 top edge "B" connections (or pins) are ground returns for the corresponding 40 lower edge "A" connections.

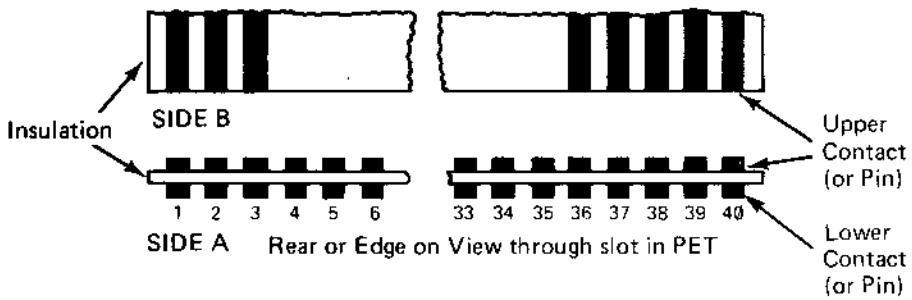


Figure 7.12. Simplified view of edge connector J4 with contact identification. All side B contacts grounded.

Table 7.13 shows the PET pin numbers, line labels and line descriptions.

Side A Pin Numbers	Line Labels	Line Description
A1	BAØ	Address bit Ø, used for memory expansion. Buffered.
A2	BA1	Address bit 1, used for memory expansion. Buffered.
A3	BA2	Address bit 2, used for memory expansion. Buffered.
A4	BA3	Address bit 3, used for memory expansion. Buffered.
A5	BA4	Address bit 4, used for memory expansion. Buffered.
A6	BA5	Address bit 5, used for memory expansion. Buffered.
A7	BA6	Address bit 6, used for memory expansion. Buffered.
A8	BA7	Address bit 7, used for memory expansion. Buffered.

Table 7.13. Memory expansion connector. PET pin numbers. Line labels and line descriptions.

Table continued on next page.

Table 7.13. Memory expansion connector. PET pin numbers. Line labels and line descriptions (continued).

Side A Pin Numbers	Line Labels	Line Description
A9	BA8	Address bit 8, used for memory expansion. Buffered.
A10	BA9	Address bit 9, used for memory expansion. Buffered.
A11	BA10	Address bit 10, used for memory expansion. Buffered.
A12	BA11	Address bit 11, used for memory expansion. Buffered.
A13	NC	No connection.
A14	NC	No connection.
A15	NC	No connection.
A16	$\overline{\text{SEL 1}}$	4K byte page address select for memory locations 1000-1FFF.
A17	$\overline{\text{SEL 2}}$	4K byte page address select for memory locations 2000-2FFF.
A18	$\overline{\text{SEL 3}}$	4K byte page address select for memory locations 3000-3FFF.
A19	$\overline{\text{SEL 4}}$	4K byte page address select for memory locations 4000-4FFF.
A20	$\overline{\text{SEL 5}}$	4K byte page address select for memory locations 5000-5FFF.
A21	$\overline{\text{SEL 6}}$	4K byte page address select for memory locations 6000-6FFF.
A22	$\overline{\text{SEL 7}}$	4K byte page address select for memory locations 7000-7FFF.
A23	$\overline{\text{SEL 9}}$	4K byte page address select for memory locations 9000-9FFF.
A24	$\overline{\text{SEL A}}$	4K byte page address select for memory locations A000-AFFF.
A25	$\overline{\text{SEL B}}$	4K byte page address select for memory locations B000-BFFF.
A26	NC	No connection.
A27	$\overline{\text{RES}}$	Reset for 6502 microprocessor. Note: connected to 74LS00 output.
A28	$\overline{\text{IRQ}}$	Interrupt request line to the microprocessor.
A29	B02	Buffered phase 2 clock.
A30	R/W	Buffered read/write from 6502 microprocessor.
A31	NC	No connection.
A32	NC	No connection.
A33	BD0	Data bit 0. Buffered.
A34	BD1	Data bit 1. Buffered.
A35	BD2	Data bit 2. Buffered.
A36	BD3	Data bit 3. Buffered.
A37	BD4	Data bit 4. Buffered.
A38	BD5	Data bit 5. Buffered.
A39	BD6	Data bit 6. Buffered.
A40	BD7	Data bit 7. Buffered.

ADDITIONAL BASIC COMMANDS

By this time, the user is probably familiar with the use of the commands INPUT and PRINT. INPUT permits the output or display of data. These commands are common to all forms of BASIC.

To add flexibility to the PET computer system, several commands have been added to classical BASIC in the PET, and future Commodore products will take advantage of the resulting extra capability. In general, enhanced flexibility of data interchange between the PET and peripheral devices is possible, thanks to the use of these extra commands.

To communicate with any device, a combination of the additional commands is used:

a)	*OPEN/CLOSE*	*Open or close logical file.*
b)	*PRINT#*	Write data from PET to I/O device.
c)	CMD	Same as PRINT# but leaves IEEE device an active listener on bus after execution of command.
d)	INPUT#	Read data from I/O device to PET.
e)	GET#	PET accepts one character from I/O device.

INPUT/OUTPUT COMMAND PARAMETERS

In order to use the additional commands referred to in the above, four parameters must be taken into consideration:

a) Logical file number (LF)
b) Device number (D)
c) Secondary address (SA)
d) File-name (FN)

These parameters can appear, for example, when using the OPEN# command in the form of the statement:

OPEN#LF,D,S,FN

LOGICAL FILES

Files are used to store and retrieve data, as for example in the case of a magnetic tape or disc file. A convenient extension of this idea is to regard any device which can receive and/or generate data as a logical file. To the PET operating system, data might just as well have come from, or be going to, a storage system such as magnetic tape.

For example, imagine that an external digital voltmeter is set up so that it can transmit voltage readings upon request to the PET via the IEEE bus. Sometime during the "voltmeter program" the programmer will have to open a file and assign a logical file number to refer to the voltmeter. Once this has been done, The PET can use a "read" command (INPUT#) which uses the logical file number to refer to the voltmeter. When no further data is required from the voltmeter, the logical file can be closed.

More generally, the advantages offered by the use of logical files are:

a) Every device number secondary address combination can be associated with its own unique logical file number within a program.
b) Multiple files within a single dev ice can be refered to by means of distinct logical file numbers. This approach is to be used in the newly developed disc storage system for the PET.
c) Once a logical file number has been defined in an OPEN

statement, within a program, only this number need be used in the following input/output statements. This eliminates the need for further restatement of device number, secondary address (where used) and file name (where used).

Although it is permissable to identify and use many logical files in a given program, the PET operating system has to keep track of the files that are currently in use in the program. The greatest number of files that can be controlled by the PET at one time is ten. Note that exceeding ten will result in loss of PET operation; this can be restored by switching the computer off and on. A logical file number can be any integer in the range 1 through 255.

DEVICE NUMBERS

All devices which the PET communicates with are assigned numbers. The first four of these are reserved for the following peripherals:

	Device Number	Device
	0	Keyboard
Default-	1	Cassette 1 panel mounted
	2	Cassette 2 add-on
	3	Video screen

All other devices are automatically assumed by the PET to be IEEE devices, and control is transferred to the device which will have been allocated a number within the range 4 through 30. Except in special cases, a specific number would be allocated to each IEEE device to allow the PET and a particular device to communicate using the parallel IEEE-488 bus.

On many IEEE devices, the allocation of the device number is made by means of a switch, or in the case of less expensive products, by the connection of jumpers.

SECONDARY ADDRESSES

The concept of secondary address may be new to those people who have never worked with the IEEE bus. The use of a secondary address permits an intelligent peripheral to function in any one of a number of modes. For example, in a PET printer, there are six secondary addresses:

	Secondary Address	Operation
Default-	0	Normal printing
	1	Printing under format statement control
	2	Transfer data from PET to format statement
	3	Set variable lines per page
	4	Use expanded diagnostic messages
	5	Byte data for programmable character

In short, by changing the secondary address used to communicate with a given physical device, its operating characteristics can be totally changed, if so desired. Many of the IEEE devices have their own particular secondary address conventions which must be followed. Specific data on these conventions can be obtained by consulting the manual for that particular device.

The PET tape units have a special set of secondary address rules:

	Secondary Address	Operation
Default-	0	Tape is being opened for "read"
	1	Tape is being opened for "write"
	2	Tape is being opened for "write" with an "end of tape" header being forced when the file is closed.

The secondary address can have values over the range 0 through 31.

FILE NAMES

In random storage devices where there is more than one file to be accessed, the use of names to identify files is mandatory. In the case of tapes, a file name is desireable, even if there is only one file on the tape, since it facilitates the identification of tapes.

For the two cassette tape units of the PET, a file name may be any combination of up to 128 characters. When a file name is searched for, it is matched on an ascending character basis.

Assume that an eight character file name COUNTING was specified when writing. If desired, this could be searched for with an abbreviated name such as COU. The first file header that is found with these three consecutive characters will then be opened and positioned on. In principle, this could include unwanted file names such as COUNT or COUNTRY, as well as COUNTING.

It is, therefore, adviseable to specify the *complete* file name in order to avoid errors.

For other devices which use named files, the individual description of the device should be consulted in order to ascertain the specific requirements for file name usage.

TAPE CASSETTE OPERATION FOR FILES

The PET devotes special attention to the two tape cassette units that can be attached to it. The tape units are specially modified so that the PET has control over the motor movement of the cassette.

It can also sense when the PLAY, REWIND, or FAST FORWARD buttons have been pushed; this is done by means of a single switch mounted in the tape unit.

Note that the same switch is used to sense all three buttons: if *any* of the three is pushed, the PET will think that you pushed the PLAY button and will respond accordingly.

Because of the type of mechanism used in the tape unit, the user must rewind, fast forward, stop, load and eject tapes. He must also put the unit into the write mode by pushing the record button either simultaneously with, or before the PLAY button is pressed.

The PET has total control over the movement of the tape once the appropriate buttons have been pushed to engage the motor.

Messages displayed throughout the program will tell the user when it is necessary for him to initiate the function of play or record. Logic dictates the times when the user should rewind and fast forward.

The two tape units of the PET are handled independently, and the various control lines permit the reading of data from cassette #1, the reading of data from cassette #2, motor control of cassette #1, motor control for cassette #2 and a common write line.

FILE RECORDING TECHNIQUE

The data structure embodied in the tape files will ensure the maximum memory utilization and maximum reliability of recording.

To accomplish this, the PET records data at two audio frequencies in two consecutive blocks of data. All of the data is totally repeated and by means of error checking techniques incorporated in the PET software, it is possible for most audio dropouts to be corrected by the operating system utilizing the redundant data stored in the second data block.

In order to correct for (a), the fact that tape units record at different speeds, and (b), the normal drag characteristics of tapes, a speed correlation technique is used during reading. To correct for the individual start/stop characteristics on the tape and syncronize correctly between each block on tape, a

single tone is written between blocks. This signal is used to syncronize both position and speed of the tape. Varying lengths of tone are used at the beginning and between the data blocks of the tape. By writing about ten seconds of the tone on each opening of a file, the PET automatically corrects for normal leader. Individual tape blocks are separated by shorter tone durations.

FILE HEADERS

An important assumption underlying the tape system design was that the user would often want to record more than one file of data on a tape. In order to facilitate this and to allow for proper label checking, the first physical data recorded on tape for any operation is a file header. This file header looks exactly the same as the normal data block, except that the first character of every block on tape contains an identification character which enables the operating system to differentiate between program blocks, data blocks, file headers and end of tape headers.

The PET allows for up to 128 characters of a file name to be stored in the file header. This is the name which is searched for and matched on in the various OPEN/CLOSE options.

TAPE BUFFERS

Another basic premise in the design of the tape operating system was that the user would want to write tape independently of what is occurring on tape at a given moment. This is accomplished in the operating system by permanently assigning a block of memory as a data buffer for each cassette. A 192 character buffer is located at decimal address 634 for cassette #1, followed by a 192 character buffer at decimal address 826 for cassette #2. The tape file header is written into the buffer first and then written on tape.

Data files are accumulated in the tape buffer until 192 characters are exceeded, then the contents are either written on tape for write, or if the program is reading tape, the next block of data is read into the buffer. Tape file headers and all data blocks are, therefore, 192 characters long.

Tape buffers are not used in the case of program files, since these are written onto the tape directly from the memory in which the program resides. In order to accomodate the variable memory location, the file header for a program file contains the beginning and ending address for the program. The full program is written onto tape in the usual form of two consecutive redundant blocks.

MULTIPLE FILES

In order to have multiple files on tape, the user needs the ability to add files to a tape and also know when a tape is at its end. It is, therefore, important that the operating system give an "end of file" and "end of tape" indication.

In the case of data files, an "end of file" marker is appended after the last data character. This is available to the user in a status word on reading; the "end of file" marker is automatically inserted when a write file is closed.

In the case of program files, because all data is always contained in a single block, the end of the block signifies the end of the program.

To signify that the end of the tape has been reached, a special separate file header is written. When this is encountered during a search for files, the PET automatically stops the tape and indicates "file not found" to the user. A typical multiple file tape could contain first a data file, then a program file, followed by an "end of tape" header as illustrated in the example of figure 7.14.

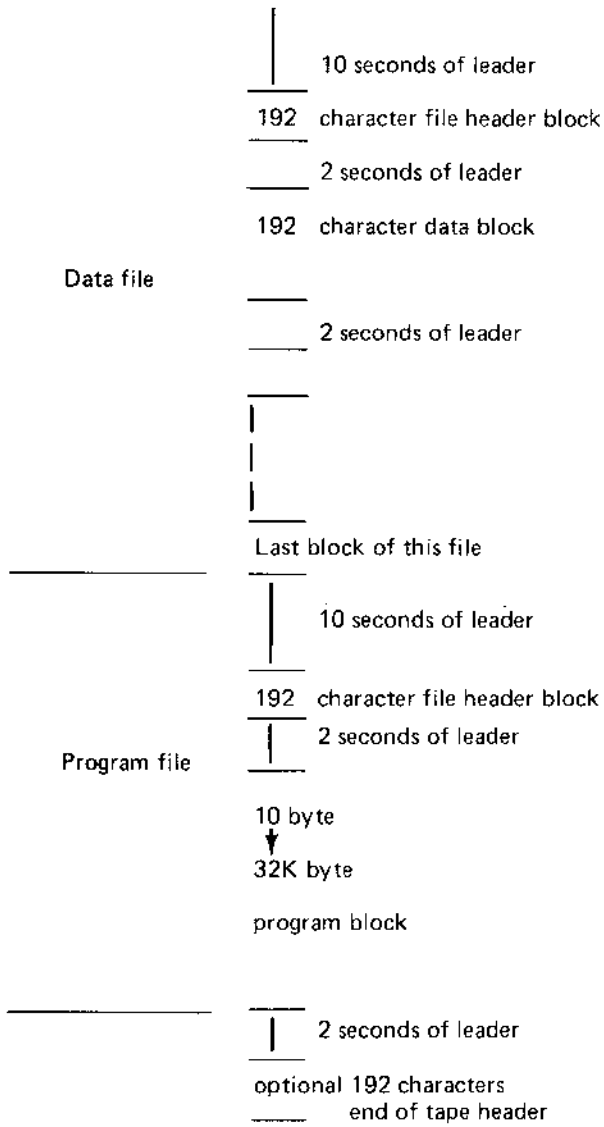


Figure 7.14. An example of multiple file structure.

LOGICAL FILE I/O OPERATIONS: GENERAL

These operations can be subdivided into three steps:

a) Open the file - tell the PET everything it needs to know about the file.

b) Read data from, or write data to the logical files.

c) Close the file - allow the PET to clear up the device and terminate the active file.

These steps are discussed in detail on the following pages.

OPENING FILES

In order to tell BASIC about the file you want to operate on, it is first necessary to open the file. This is done by the following statement:

OPEN logical file, device, secondary address, file name

More specifically, the statement consists of the command OPEN followed by the logical file number, then the device number to which the file is assigned, then the secondary address data (if any) communicated during the interaction of BASIC with the file, and last, the name of the physical file (if any).

This statement, or expression, is interpreted by BASIC, and could, therefore, use *computed* logical file numbers, device numbers or secondary address data. This capability is extremely useful when handling multiple file devices such as discs.

The keyword OPEN and the logical file numbers are essential in order to open a file; that is address a device in preparation for a "read" (INPUT #) or a "write"(PRINT #).

The device number is optional; if not entered, the default value "1" will be used.

A file name is optional, though preferred, for the tape units: however, a name would be essential for a disc storage unit.

EXAMPLES OF OPEN STATEMENTS

The statement OPEN 1,2,1 is interpreted by the operating system as saying:

Parameter	
(LF)	Logical file #1 has been opened
(D)	Logical file #1 has been assigned to tape unit#2
(SA)	Tape unit #2 has been instructed to write on tape
(FN)	A file name has not been assigned to the tape record

Similarly, OPEN 3 is interpreted as saying: (F)

Parameter	
(LF)	Logical file #3 has been opened
(D)	Logical file#3 has been assigned to tape unit #1 (default "1")
(SA)	Tape unit #1 has been instructed to read from tape (default "0")
(FN)	No file name referred to

If a PET printer is assigned "4" as a device number, then OPEN 12,4,1 is interpreted as:

Parameter	
(LF)	Logical file #12 has been opened
(D)	Logical file #12 has been assigned to device #4
(SA)	Printer has been instructed to print under format statement control
(FN)	File name not applicable

Note: The PET has a special system with OPEN for tape files. The opening of the tape is automatic, but the tape header may not always be written at the beginning of the tape buffer; this implies that the operating system does not always correctly initialize the buffer point. For consistent and reliable operation of the tape file header, the following statements should be used:

1) For tape #1: POKE 243,122
   POKE 244,2
2) For tape #2: POKE 243,58
   POKE 244,3

These should be written prior to each OPEN for write.

LOAD

A special case of the OPEN command is the LOAD of a named file: a LOAD is done with the following statement:

LOAD name, device number

The operating system automatically generates an OPEN using the appropriate secondary addresses for "load". This OPEN causes the loading device to search for a program name. After the program is found, it is automatically read from the device and loaded into memory starting at an address specified in the file header. Most reading errors on the first pass through that program are automatically fixed on the second pass.

At the end of the load cycle, a checksum error, of the total program is made. If a checksum error, or if an

uncoverable read error occurred, the operating system automatically prints ?LOAD ERROR and stops the load program.

If the program load was from direct mode, the clear function is performed at the end of the load, thereby initializing all variables.

If the LOAD is called from a program, then the PET treats this LOAD as an overlay. The new program is loaded into the space used by the previous program, but the values of all of the variables are maintained from the previous program. This allows for one program to call another and pass parameters to the called programs.

The only restriction on this is that all the called programs must fit in the same, or less space as the first program.

Because BASIC totally replaces the current program, it is not directly possible to have a single main program and several subroutine overlays, however, by including the main program with each overlay, the same effect is obtained with some loss of speed.

The combination of the use of named files and overlays allows the writing of very large structured programs of appreciable complexity.

VERIFY

This very instruction is a special case of LOAD. It should be used after every program SAVE.

The command causes BASIC to go through all the steps of a program LOAD, with the exception that the data does not get loaded into memory, but, instead, gets compared with memory. If either first or second pass errors occur, the PET will type out ?VERIFY ERROR which means that the program should be saved again before it is lost. On VERIFY, the status word has the following meanings

Code	Meaning
4	Short block
8	Long block
16	Checksum error on tape
32	Checksum ERROR on tape

SAVE

SAVE also performs an automatic open and close. The SAVE is specified by the statement:

SAVE name, device number

If the physical device is one of the two tape units, the operating system automatically initiates a tape header and opens a tape file with the appropriate name. The file header is written with the beginning and ending address.

If the device is an IEEE-488 device, a special open message is sent indicating that the PET is sending a program file.

The program is then written directly from its memory locations to the tape or the IEEE-488 bus.

If the SAVE is on tape, a checksum is computed and also saved and then the whole program is written again to give the redundant recording. At the end of the program, the tape is automatically stopped and positioned for the next data.

IEEE-488 SPECIAL FEATURES

In the tape, the program beginning and ending address are stored in and retrieved from the tape file header.

In order to more efficiently use the IEEE-488 data, the starting address of the program is sent as the first two bytes of data on a SAVE and retrieved from those positions on a LOAD.

IEEE-488 OPEN CONSIDERATIONS

If the OPEN command selects a device which has a value of 4 or more, the operating system assumes that the device is an IEEE-488 device.

If the OPEN does not specify a file name, then nothing is communicated on the IEEE-488 bus. However, if a file name is specified, the operating system sends a listen attention sequence to the device number specified in the OPEN along with a secondary address which is the OR of hexadecimal "F0" and the secondary address specified in the OPEN statement.

Commodore-supplied peripherals, such as the floppy disc storage system, will use this secondary address and also the file name, which is then transmitted to the listening device in order to transfer data later to the open file.

TAPE FILE OPERATION MODES

tape files can be opened for two distinct purposes:

a) In order to write from the PET onto tape.
b) In order to read from tape to the PET.

OPEN FOR WRITE ON TAPE FROM PET

The flow diagram of Figure 7.15 outlines the PET-user interaction and PET function when opening a file for write on tape. The initial block shows that there are two ways of opening the file:

a) OPEN for write-data tape.
b) SAVE-write a program tape.

Note that if the tape file is opened directly from the keyboard, then the message WRITING NAME is displayed. If the file is opened under program control, and the PLAY and RECORD buttons are depressed previously, then no message appears on the screen. In this manner, any display material placed there by the current program is not disturbed.

OPEN FOR READ FROM PET TO TAPE

The flow diagram of Figure 7.16 outlines the PET-user interaction and PET function when opening a file for reading on tape. The initial block shows that there are two ways of opening the file:

a) OPEN for read data tape.
b) LOAD program into memory.

Note that if the file is opened directly, that is from the keyboard, then the messages PRESS PLAY, SEARCHING FOR NAME and FOUND NAME are displayed. If LOAD was used, then the BASIC variables of the loaded program are initialized.

If the file is opened under program control and provided that the PLAY button had been pressed previously, no messages appear on the video screen in order to disturb material displayed by the current program. Initialization of the BASIC variables does not occur.

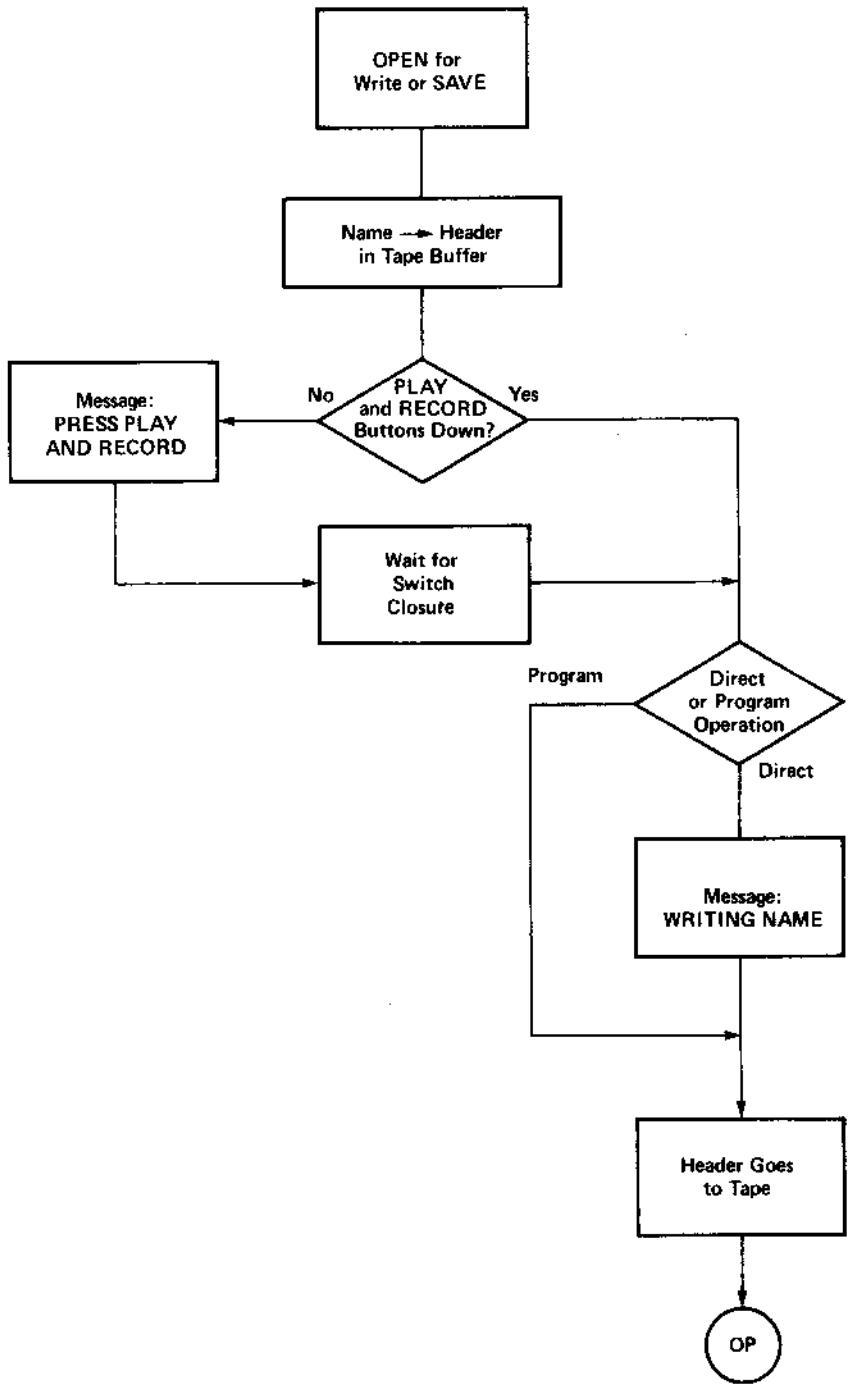


Figure 7.15. OPEN for write from PET: PRINT#,CMD or SAVE.
OP = operating system takes over.

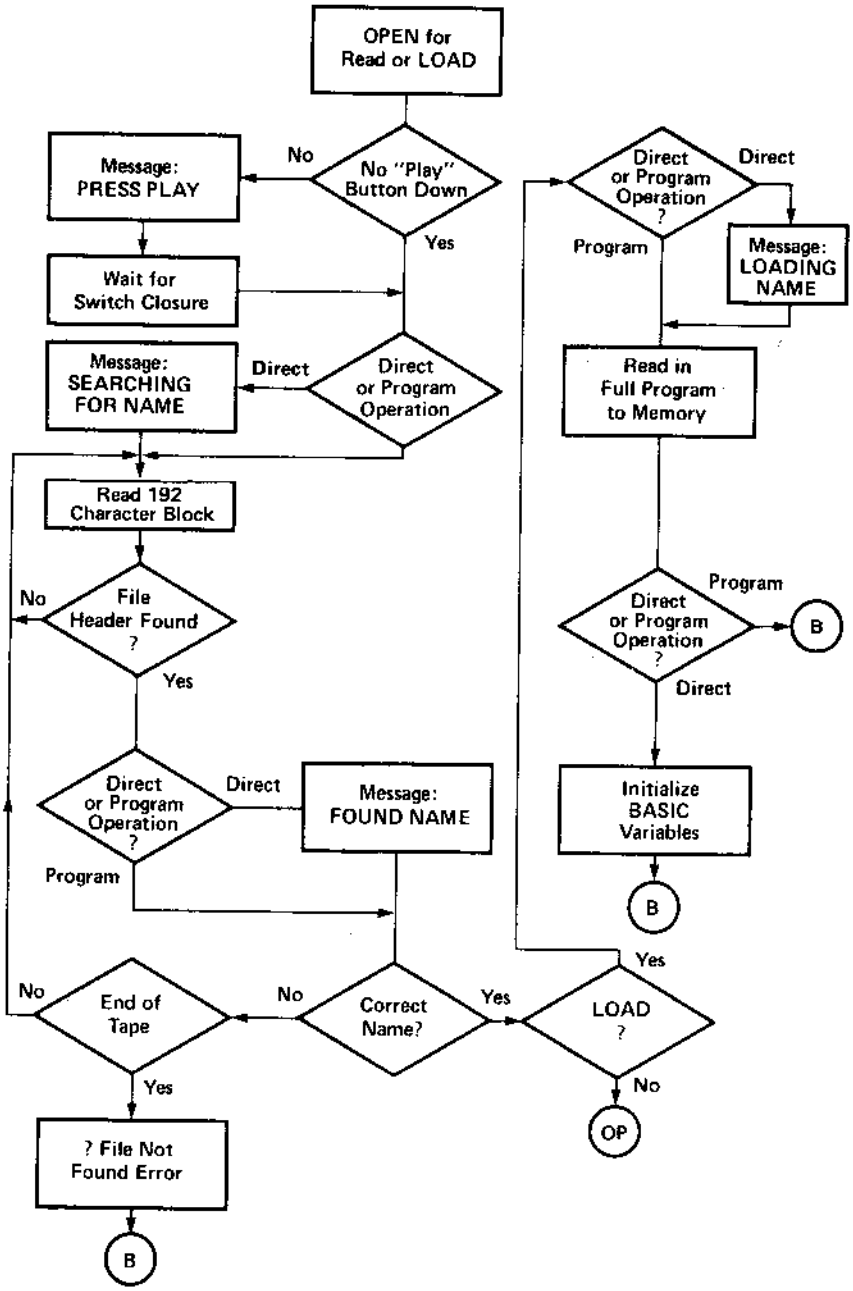


Figure 7.16. OPEN for read to PET: INPUT# or LOAD
OP = Operating system takes over. B = BASIC takes over.

DATA INPUT: GENERAL

The use of the word "input" in this context implies input of data *to* the PET *from* any device.

INPUT#-String and Variable Input

INPUT# is the command used to initiate data transfer from I/O devices to the operating system. The statement format is:

INPUT# logical number file, A,A$,B,B$,etc.

Where A,A$,B, and B$ are numerical and string variables to be inputted (read) from the selected logical file to the operating system one character at a time.

Because the rules for the BASIC interpreter apply to these input statements, all carriage returns, commas, terminate fields, nulls, preceeding blanks (except in strings), and other control characters are automatically deleted.

It is not always possible to mix both numeric and alphabetic data on the I/O device. If a numeric field is specified, only numeric data in the standard form expected by BASIC is accepted, otherwise a ?BAD DATA ERROR meesage is displayed.

If there is any ambiguity about the data coming in, the user should input only to strings and then use the various string manipulation commands to process the data into the appropriate variables.

Example of Input# *Statement*

If X represents a series of 50 numbers stored on a tape file named VECTOR and we assume that the PLAY button has just been depressed on the tape unit#1. Then the following program will read the 50 numbers one at a time and display them on the video screen.

```
10 OPEN 1,1,0"VECTOR"     Open logical file #1. Assign file to cassette 1. Open tape for
                          "read". Look for physical file named VECTOR.
20 FOR K = 1 to 50        Read 50 numbers at one time from cassette 1.
30 INPUT#1,X
40 PRINT X                Display numbers on video screen
50 NEXT K
60 CLOSE 1                When 50 numbers have been read, close logical file #1.
```

GET #-CHARACTER TRANSFERS

Not all devices transfer data in a form which is accceptable directly to BASIC. There is a series of binary data and combinations which BASIC ignores and although many IEEE devices do correctly respond with data formats which are acceptable to basic, not all do.

In addition, in some cases, it is desirable for the programmer to have immediate access to characters as they are transfered to the system. GET- fetches from the IEEE-488, or tape device, a single character at a time, putting a character in a field specified following the GET#. THE FORM IS:

GET# logical file, field

TAPE INPUT

When reading from the tape file, the data comes to the user I/O independent. Each time BASIC starts on INPUT# *or GET#* from a logical device which was opened for read on tape 1 or 2, a special subroutine is called, which initiates tape input.

As each character is requested from BASIC, it is fetched from the appropriate tape buffer. When the buffer is empty, the tape input routine suspends the user program and reads the data block from tape into the buffer and then transfers the next character to BASIC. If a read error occurs, it is noted in the

status word.

When the end of file mark is encountered in the buffer, the end of file position of the status word is set on and carriage returns are forced automatically out until the command is finished.

At the end of a command, BASIC calls another routine which reinitializes the input to be the keyboard and tells the end of file operation that a command is complete.

IEEE-488 DEVICE INPUT SEQUENCES

All INPUT# or GET# commands go through the same sequence. When the command is first encountered, the IEEE-488 input initiation routine is called, which sends a talk attention sequence to the device and secondary address which was specified for that logical file in the OPEN sequence. At the end of the attention sequence, the PET establishes itself in a listener mode and attempts to wait for a DAV signal indicating a single character has been received. If the DAV is received within 65 milliseconds, that character is handed to BASIC and/or to the other program calling the IEEE-488 routine. Each time the IEEE-488 routine is called, it will go through the same sequence of getting a single character while waiting for a time out to occur. If the bus does not respond in 65 milliseconds, then the IEEE-488 routine will automatically terminate the sequence; giving a read error in the status word to indicate that it has terminated the sequence.

If during the course of reading the character, the IEEE-488 routine senses an EOI line, it will indicate the end of information in the status word and will continue to return carriage returns, until the command it has been currently operating under has been terminated. At the end of the command, BASIC calls a termination subroutine which reinitializes the device to the keyboard and sends an untalk to the IEEE-488 bus, thereby, freeing the bus for the next command.

INPUT BUFFER LIMITATIONS

Although data is transferred from the operating system one character at a time, in order to edit, BASIC accumulates these characters into an 80 column input buffer. This buffer must be terminated by a carriage return.

On the PET, should more than 80 characters be read, the operating system will malfunction, as the operating system variables are overwritten. The PET can be made to function again by switching the line supply off and on.

This constraint must be kept in mind when using tape and disc file systems.
This means that carriage returns must be written on tapes, discs, or other I/O devices in such a way that not more than 80 characters per field are written without being separated by carriage returns.

If an I/O device sends more than 80 characters, the GET command can be used to build your own string without running into the buffer limitation.

DATA OUTPUT: GENERAL

The use of the words "print" and "write" refers to data output *from* the PET *to* any device.

PRINT#

The command PRINT# must be followed by a logical file number, and then a comma to separate the data that would follow PRINT:

PRINT# logical file number, data

Data is transferred a single character at a time to the physical device correlated with the logical file specified in the relevant OPEN statement. Many of the file delimiters such as commas are automatically

deleted by BASIC; although this does not greatly effect the printing, it should be remembered that when reading back from tape or another I/O device that file delimiters must be forced. This forcing can be done by inserting a CHR$(44) or "," between fields or by only printing single fields in each PRINT# statement which will force carriage returns between fields. Example:

instead of writing

```
PRINT#LF,A;B$;C$
```

which will be sent as

```
AB$C$
```

with no delimiters:

```
PRINT#LF,A;CHR$(44)B$;CHR$(44);C$
```

or:

```
PRINT#LF,A";";B$;";";C$
```

which will output: (Note: CR means carriage return)

```
A,B$,C$,CR
```

or:

```
PRINT#LF,A
PRINT#LF,B$
PRINT#LF,C$
```

which will output:

```
A CR B$ CR C$ CR
```

Because BASIC always formats outputs to any devices as though it were outputting to the screen, PRINT#LF,A,B has several skip characters between the values of A and B, while A;B does not have any extra skips.

An exception to this rule is the tape where the first skip on output is supressed.

Note: Although both the INPUT# AND PRINT# commands operate in virtually the same way as their equivalent INPUT *and* PRINT *statements do in BASIC, the abbreviated command ? which can be used in place of* PRINT, *does not apply to* PRINT#. ?# *and* PRINT# *are recognized and reduced to two different token characters when processed by BASIC. ?# will look like* PRINT# *when listed but gives* ?SYNTAX ERROR *when an attempt is made to execute it.*

Examples of the PRINT# *Statement*

This program will print the series of numbers 1,2,3...50, one at a time on a PET printer.

```
10OPEN 5,4,0        Open logical file #5. Assign logical file #5 to device #4 (PET
                    printer) in normal print mode corresponding to secondary
                    address "0".
20 FOR K = 1 to 50  Print the series of 50 numbers on printer.
30 PRINT#5,K
40 NEXT K
50 CLOSE 5          Close logical file #5.
```

To write the above series of numbers on a cassette in tape unit #2, only the OPEN line would have to be modified, if the same logical file numbers were chosen:

```
10OPEN 5,2,1        Open logical file #5. Assign logical #5 to device #2 (tape unit
                    #2) with a write without "end of tape" designation
                    corresponding to secondary address '1'.
```

```
20 FOR K = 1 to 50        Record the series of 50 numbers on tape.
30 PRINT#5,K
40 NEXT K
50 CLOSE 5                Close logical file #5.
```

In the above cassette example, the data would be accumulated in a 192 character buffer one character at a time. When the capacity of the buffer is exceeded, then data entry is suspended, the tape started, and the buffer contents written to tape. The buffer is initialized to accept up to 192 characters and then the program is allowed to proceed.

Note: Not all tape units currently operate with the same START/STOP *characteristic as defined for the original tape operating system. In order to obtain reliable operation of the tape recorders, the 192 characters of the buffer should be monitored by the program. Prior to transferring 192 characters, the programmer should turn on the appropriate cassette motor and then wait for at least .1 second before transferring the last character.*

There are several ways to accomplish this. The simplest is to just POKE 59411,53 *for cassette #1 and* POKE 59456,207 *for cassette #2 after every* PRINT *statement, this keeps the motor on all of the time and eliminates the problem.*

On the other hand, if your programs have time consuming functions like human input, sorting, or other long program run times, you should not run the motor all the time, but obtain the delay either putting a delay loop before each print, or turning the motor off with a POKE 59411,61 *for cassette #1 or a* POKE 59456,223 *for cassette #2 before the long function and turning it back on after it.*

IEEE-488 BUS OUTPUT

The PRINT# command causes BASIC to call an output subroutine which initializes an IEEE-488 device for output. The first step in the command is that the PET reassigns its normal output from the screen device to the physical device that was chosen for the logical file in the open routine. A listen command is sent on the IEEE bus to the physical device and a secondary addressd specified for that logical file in the OPEN.

BASIC then hands one character at a time to another subroutine which proceeds to transfer that character over the bus with the PET acting as a talker and all addressed devices responding listeners.

When BASIC has finished the PRINT#, another subroutine in the operating system is called and the PET sends an "unlisten" command to the entire bus and restores the primary address to the screen. This frees the whole bus for the next operation.

This unlisten sequence also sends an EOI signal on the bus, along with the last character sent from BASIC. To accomplish this, each character is stored in a buffer prior to transmission by the IEEE routines and the previous character is sent.

CMD COMMAND

Normally, each print command deals only with one logical device and at the end of the command entire bus is unlistened. In some instances, it is advisable to have more than one device on the bus; in order to facilitate this, the special command CMD is provided. CMD is virtually identical to PRINT#, except that at the end of the data transfer, the unlisten routine is not called, thereby leaving the device on the bus as a listener.

The operating system continues to treat the last device to be commanded by the CMD as the primary output device for BASIC. PRINT or LIST commands are then directed to this primary device, rather than to the video screen. More specifically, the CMD of the printer device, followed by LIST, results in hard copy

printed listing, instead of a video screen listing. However, since neither the CMD nor LIST command terminate bus operation for the device, a PRINT# is required to terminate a CMD command.

Examples of a CMD *Command*

To list:

```
OPEN 3,4          where 4 is the printer device number
CMD 3
LIST              will list just the same as the screen, except on the printer.
```

to print and write a disc at the same time:

```
*CMD 3            where logical file 3 is open to the printer.
PRINT#15,A,B,C    where 15 is the floppy disc logical file number
                  (previously opened).
```

will result in A,B, and C being stored on the floppy but also being displayed on the printer.

To monitor an input device:

```
**CMD 3           turn on printer
INPUT#15,A,B,C    read from floppy
```

This will result in the data from the floppy being transferred to A, B and C but also being printed as they are being transferred.

CLOSING FILES

Any logical files which have been opened during a program should preferably be closed when no longer required, and in the case of tape or disc files, must be closed before the program ends. The following should be kept in mind:

a) If the total number of logical files currently exceeds ten, then loss of PET operation will result.

b) If a logical file assigned to a tape unit is not closed, no "end of file" mark will be recorded at the end of the physical tape file. If this tape is then loaded into memory, the PET will have no way of knowing the file has ended, and if the unwanted and erroneous data is present from a previous recording, it will also be read into memory.

EXAMPLE OF A CLOSE STATEMENT

To close any file, the following simple statement is sufficient:

CLOSE logical file

If it is required to close logical file number 5, then this becomes:

CLOSE 5

TAPE FILE CLOSURE

If a file had been opened on the tape, there are two operations that occur: an "end of file" marker is recorded in the next data character, then the tape buffer is forced out onto the cassette.

If during OPEN the "end of tape" option was chosen, an "end of tape file" header block is also forced out on the cassette.

---

*Must be given each time because PRINT# unlistens the bus.

**Need not be given each time, more code can be included between instructions.

IEEE-488 NAMED DEVICE CLOSURE

For IEEE-4888 devices, which were opened with file names, a special listener command sequence, with the special secondary address of thehexadecimal E0 OR'ed with the secondary address from the OPEN is sent. This allows devices such as disc files to be closed by the peripheral controller.

ERROR DETECTION: GENERAL

The basic concept of the PET operating system is that the user will be allowed to operate in a free-form format; reading and writing on tapes, discs, and printers, in the manner that is most comfortable for him. Because I/O is totally free-form, it is most important that the operating system should have means of informing the user when transmission errors or end of data conditions occur.

STATUS WORDS

In order to facilitate INPUT/OUTPUT operation error detection, the PET uses the "status word" concept in which a byte in memory is manipulated by each of the I/O operations for the PET, and can be sampled by the programmer at any time by calling the function ST. Each bit in the staus word has a general meaning for all operations and a specific meaning for the individual I/O device.

Table 7.17 shows the errors as a function of the ST word value for the tape cassette units. IEEE read/write operations, tape verify and load operations.

ST Bit Position	ST Numeric Value	Cassette Read	IEEE R/W	Tape Verify + Load
0	1		Time out on write	
1	2		Time out on read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	-128	End of tape	Device not present	End of tape

Table 7.17. Status Word (ST) values correlated with tape cassette, unit and IEEE bus read/write errors.

IEEE DEVICE ERRORS

There are basically three errors that can occur during an IEEE-488 transfer. First, the entire bus does not respond to an attention sequence. If this occurs, the IEEE-488 subroutine sets the DEVICE NOT PRESENT bit (7 or -128). The PET also terminates the current program with ?DEVICE NOT PRESENT ERROR. If the bus responds correctly to the attention, but when the PET goes to write the first character to the bus and the physical device is not present as indicated by having NRFD or NDAC low, the PET, again, gives a device not present indication.

The second error occurs during the process of transferring data to the device. The bus does not respond

in the appropriate times and/or if it ceases to respond by means of bringing NRFD and NDAC both high, a write error indication is given in bit 0.

The third error occurs when during read on an IEEE-488, the IEEE device has not sent DAV in less than 65 milliseconds; bit 1 of the status word is then set. Whenever the EOI line is encountered, the subroutine sets the bit 6 on in the status word and continues to force carriage returns.

TAPE UNIT ERRORS

The cassette only checks data on read. The errors deleted are:

1) SHORT BLOCK (4).When reading a block from tape, a spacer tone was encountered before the expected number of bytes has been read from that block. Possible cause: attempting to read a short load file as a data record.

2) LONG BLOCK (8).When reading a block from tape, a spacer tone was not encountered after the expected number of bytes had been read from that block. Possible cause: reading a long load file as data.

3) UNRECOVERABLE READ ERROR (16).Cause: more than 31 errors on the first block of redundant blocks-or an error that could not be corrected because it occured in the same place in both blocks.

4) CHECKSUM ERROR (32).After a LOAD or reading of data, a checksum is computed over the bytes in RAM and compared to a byte received from the input device. If they do not match, this bit is set.

5) END OF FILE (64).This bit is set when the end of data file mark is encountered in a tape record.

6) END OF TAPE (-128).An EOT record was read.

EXAMPLES OF ST USE

As you can see, there is no status that the PET detects for the writing of tapes, nor errors detected for printing to and reading from the screen. There is an error on writing data out to the IEEE-488 and there is also a series of errors detected on inputting from the IEEE-488 or from tape.

The normal programming technique is to follow INPUT# or a GET# by either a test or storage of the value of status. As this is only a single byte of memory and the status changes on each new I/O command, the status is very transient.

```
100 INPUT#2,A
110 INPUT#5,B
120 IF ST=0 THEN 200
```

This code only checks the result of the transfer of data from logical file 5. The results of reading logical file 2 is forever lost. Similarly:

```
100 INPUT#2,A
110 PRINT A
120 IF ST=0 THEN 200
```

In this case, the ST reflects the print status, rather than the results of reading #2.

A correct way to use ST is the folllowing:

```
100 INPUT#2,A,B,C
110 IF ST=0 THEN 200          process normally
120 IF ST=64 THEN 300         end of data processed with no errors
```

```
130 IF ST = 2 THEN 400          time out with no errors
```

Each error can now be processed with the following:

```
140 IF ST AND mask THEN         Mask represents the bit being tested
```

POLLING TECHNIQUES

One technique to poll slow IEEE-488 devices such as sampling devices, which take many minutes to respond, is to use the INPUT# from the device; then if the status indicates time out, process other routines or loop on the INPUT # until no error occurs. If there are no errors, the correct data has been finally read and one can process that data information.

By using this sampling technique, a whole series of slow devices can be serviced, along with running a foreground program by use of the real time clock (TI,TI$) and the INPUT#/timeout error sequence, to occassionally poll devices.

DEFAULT PARAMETERS

Parameter	Default Value	Default Operation
Device #	D=1	Cassette #1 selected
Secondary address	SA=0	On tape files open for read On IEEE-488 devices, no secondary address is sent.

Table 7.18. Default values.

Statement	Equivalent (Default) Parameter Values	Operation
OPEN 1	OPEN 1,1,0	Open logical file #1 for cassette #1 read no file name
OPEN 1,2	OPEN#1,2,0	Open logical file #1 for cassette #2 read no file name
OPEN 1,2,1	OPEN#1,2,1	Open logical file #1 for cassette #2 write no file name
OPEN 1,2,1, "DAT"	OPEN#1,2,1, "DAT"	Open logical file #1 for cassette #2 write file named "DAT"

Table 7.19. Example of default parameters.

INTRODUCTION TO THE IEEE-488 BUS

This bus consists of 16 signal lines that are divided functionally into three groups, those are:

a) The data transmission bus
2) The control bus
3) The management bus

Furthermore, the IEEE bus can support three classes of device:

a) Talkers: at any given moment, only one device is permitted to transmit data to the data bus.
b) Listeners: as many devices as required may receive data simultaneously from the bus.
c) Controller: the PET is the *only* controller allowed on the IEEE bus.

BUS/DEVICE CONTROL

The line-pin connections for the 12 position, 24 contact edge card connector, emanate from the PET main assembly board (see Table 7-19). For further information, please refer to Figure 7.2

Certain physical limitations should be noted when connecting devices to the IEEE bus:

a) The maximum advisable bus extension from the PET is 20 meters.

b) The maximum interdevice spacing is 5 meters.

c) The maximum number of devices is 15.

PET Contact Identification	Bus	IEEE Label	PET Contact Identification	Label Description
1	DATA	DI01	1	Data INPUT/OUTPUT LINE #1
2		DI02	2	Data INPUT/OUTPUT LINE #2
3		DI03	3	Data INPUT/OUTPUT LINE #3
4		DI04	4	Data INPUT/OUTPUT LINE #4
5	MANAGER	EOI	5	End or identify
6	TRANSFER	DAV	6	Data valid
7		NRFD	7	Not ready for data
8		NDAC	8	Data not accepted
9	MANAGER	IFC	9	Interface clear Same as PET reset
10		SRQ	10	Service request
11		ATN	11	Attention
12		SHIELD	12	Chassis ground and IEEE cable shield
A	DATA	DI05	13	Data INPUT/OUTPUT LINE #5
B		DI06	14	Data INPUT/OUTPUT LINE #6
C		DI07	15	Data INPUT/OUTPUT LINE #7
D		DI08	16	Data INPUT/OUTPUT LINE #8
E	MANAGER	REN	17	Remote enable (REN) always ground in the PET
F	GROUNDS	GND6	18	DAV ground
H		GND7	19	NFRD ground
J		GND8	20	NDAC ground
K		GND9	21	IFC ground
L		GND10	22	SRQ ground
M		GND11	23	ATN ground
N		LOGIC GND	24	Data ground (DI01-8)

Table 7.20. IEEE bus group, label and contact identification number.

THE DATA BUS

This bus is comprised of 8 bi-directional lines that transmit the active low data signals D101-8. The slowest device in use on the bus at a given time controls the rate of data transfer; the mode of transfer is one byte at a time, bit parallel.

Peripheral addresses and control information are also transmitted on the data bus. They are differentiated from data by ATN (true) during their transfer.

The most significant bit (MSB) is on line D108.

For an explanation of signal abbreviations such as D1-08, see Figure 7.23.

Data Transmission Modes

All possible bit patterns are valid on the data bus when sending data to devices.

THE TRANSFER BUS

This three line bus controls the transfer of data over the data bus. The signals transmitted are used in

the handshake procedure outlined in 7-21.

These signals are:

a) NRFD Not ready for data
b) NDAC Data not accepted
c) DAV Data valid

Note that the talker originates the DAV signal and the listeners the NFRD and NDAC signals. See Table 7-23 for detailed description of signals.

The Handshake Procedure

When a talker transmits a data byte to one or more listeners, this control procedure is used in order to ensure that the operation is successful.

The essential function of the handshake is to ensure;

a) All listeners are ready to accept data.
b) That there is valid data on the data bus.
c) That the data has been accepted by all listeners.

The transfer of data occurs at a rate determined by the slowest active device on the bus; this allows the interconnection of devices which handle data at different speeds.

The sequence of events that occur during the transfer of a data byte from the talker to the listeners is shown in the flow diagram of figure 7-21.

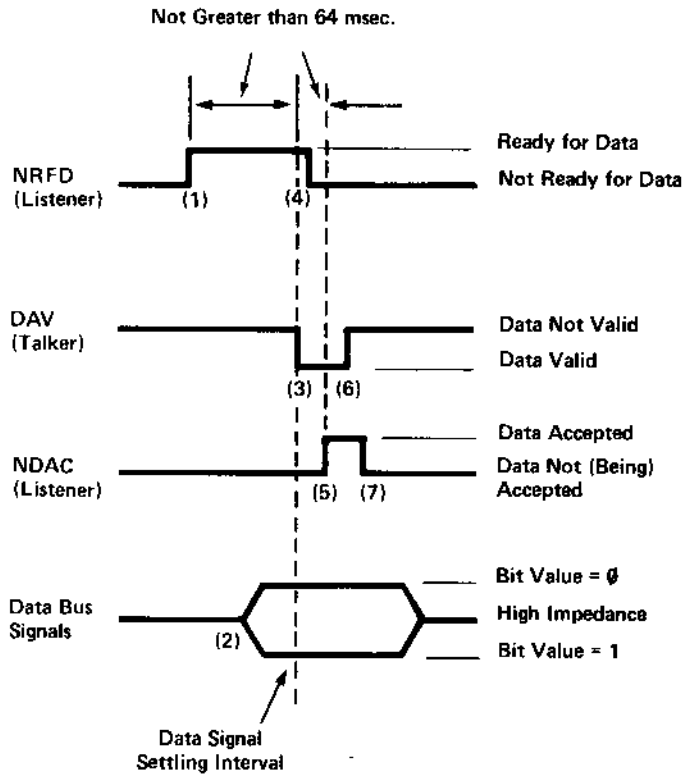


Figure 7.21. Transfer bus handshake sequence.

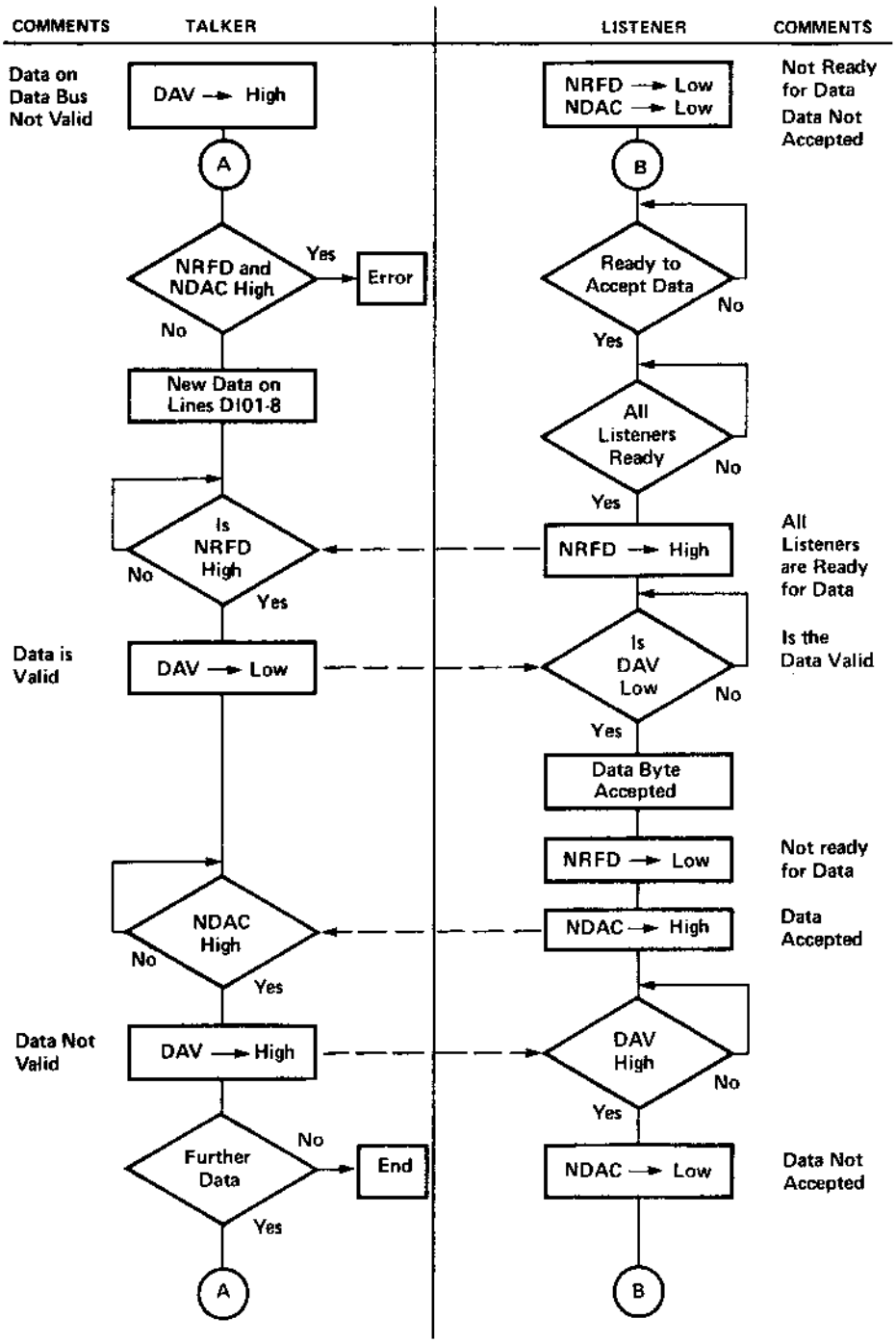


Figure 7.22. Sequence of events during a data byte transfer from the talker to the listners. Broken lines indicate the testing of transfer bus signal logic levels.

Figure 7-22 shows the relative timing of transfer bus signals during a typical handshake; the bracketed numbers in the following sequence refer to the changes in signal logic levels in the Figure:

1) NRFD goes high (false) indicating that all listeners are ready for the next byte of data.
2) The talker puts the next data byte on the data bus and allows the data signals to settle. This could happen before, after or during (1).
3) The talker tests NFRD, when it is found to be too high, the talker makes DAV low (true) to inform listeners that the bus data is now valid.
4) As soon as a single listener detects that DAV is low, that listener sets NRFD low; data is now accepted by all the individual listeners at their own rate, each of whom release NDAC as they accept the data.
5) NDAC goes high (false) when the slowest of the listeners have accepted the data.
6) The talker sets DAV high (false) indicating that the bus signals are now invalid.
7) The listeners note that DAV has gone high and sets NDAC low (true) completing the handshake. When each listener has processed the data, they release NFRD. This terminates the sequence for the first data transfer. The sequence will repeat again, beginning at (a), until all required data transfers have been completed.

PET/IEEE Bus Timing Constraints

The following limitations should be noted in order to avoid a loss of data:

a) When PET is a listener, it expects DAV to go low within 64 milliseconds after it has set NFRD high.
b) When PET is a talker, it expects NDAC to go high within 64 milliseconds after it has set NRFD high.

If these limitations are exceeded, the PET ceases to transfer and sets the appropriate status word (ST). See Table 7-24.

THE MANAGEMENT BUS

This group of five signal lines controls the state of the data bus and defines its signals; these can be concerned with data, addresses, or control information (device commands).

The five management signals are:

a)	ATN	Attention	Assigns devices to act as listeners or talkers.
b)	EOI	End or identify	Indicates that the last data byte is being transferred.
c)	IFC	Interface clear	Initializes the data bus. Talkers and listeners set idle. Same signal as reset in the PET.
d)	SRQ	Service request	Device tells controller that service is required. Not implemented in BASIC but available in PET.
e)	REN	Remote / enable	Permanently tied to ground in the PET.

IEEE SIGNALS AND DEFINITIONS

The 16 transmission lines of the IEEE-488 bus are each assigned a specific signal. Table 7-23 gives the bus group, name, abbreviation and functional description for each of these signals.

LOGIC LEVEL CONVENTION

The "true" or logical "1"is low with common collector type outputs. This allows any device to hold the bus in the "true" or logical "1" state.

Bus Group	Signal Abbrev.	Name	Functional Description
Manager	ATN	Attention	The PET (controller) sets this signal low while it is sending commands on the data bus. When ATN is low, only peripheral addresses and control messages are on the data bus. When ATN is high, only previously assigned devices can transfer data.
Transfer	DAV	Data Valid	When DAV is low, this signifies that data is valid on data bus.
Manager	EOI	End or Identify	When the last byte of data is being transferred, the talker has the option of setting EOI low. The PET always sets EOI low while the last data byte is being transferred from the PET.
Manager	IFC	Interface Clear	The PET sends its internal reset signal as IFC low (true) to initialize all devices to the idle state. When PET is switched on or reset, IFC goes low for about 100 milliseconds.
Transfer	NDAC	Data Not Accepted	This signal is held low (true) by the listener while reading. When the data byte has been read, the listener sets NDAC high. This signals the talker that data has been accepted.
Transfer	NRFD	Not Ready for Data	When NRFD is low (true), one or more listeners are not ready for the *next* byte of data. When all devices are ready, NRFD goes high.
Manager	SRQ	Service Request	Not implemented in BASIC, but available to the PET user.
Manager	REN	Remote Enable	REN is held low by the bus controller. The PET has a pin grounded that keeps REN permanently low.

Table 7.23. IEEE-488 bus signal.

Table continued on next page.

Table 7.23. IEEE-488 bus signal (continued).

Bus Group	Signal Abbrev.	Name	Functional Description
Data	DI01-8	Data input/ output lines 1 through 8	These signals represent the bits of information on the data bus. When a DI0 signal is low, it represents 1 and when high Ø.
General	GND	Ground	Ground connections: There are six control and management signal ground returns, one data signal ground return and one chassis shield ground lead.

STATUS WORD (ST)

ST is a BASIC variable which can be used to check the outcome of INPUT/OUTPUT operations. ST can have certain values over the range -128 to 127. Table 7-24 shows the status code that appertains to the IEEE-488 bus.

ST	Error	Explanation
1	Time out on listener	The IEEE device has not responded within the 65 milliseconds time out inverval.
2	Time out on talker	The IEEE device has *not* provided an active "data valid" signal (DAV low) within the 65 millisecond time out interval.
64	End or identify (EOI)	EOI has gone low (true), on the last byte of data being transferred on IEEE bus. Note that all devices do not generate an EOI signal. Consult relevant instrument manual.
-128	Device not present	Device did not respond when addressed; this generates an error message and the operating system returns the PET to BASIC command level.

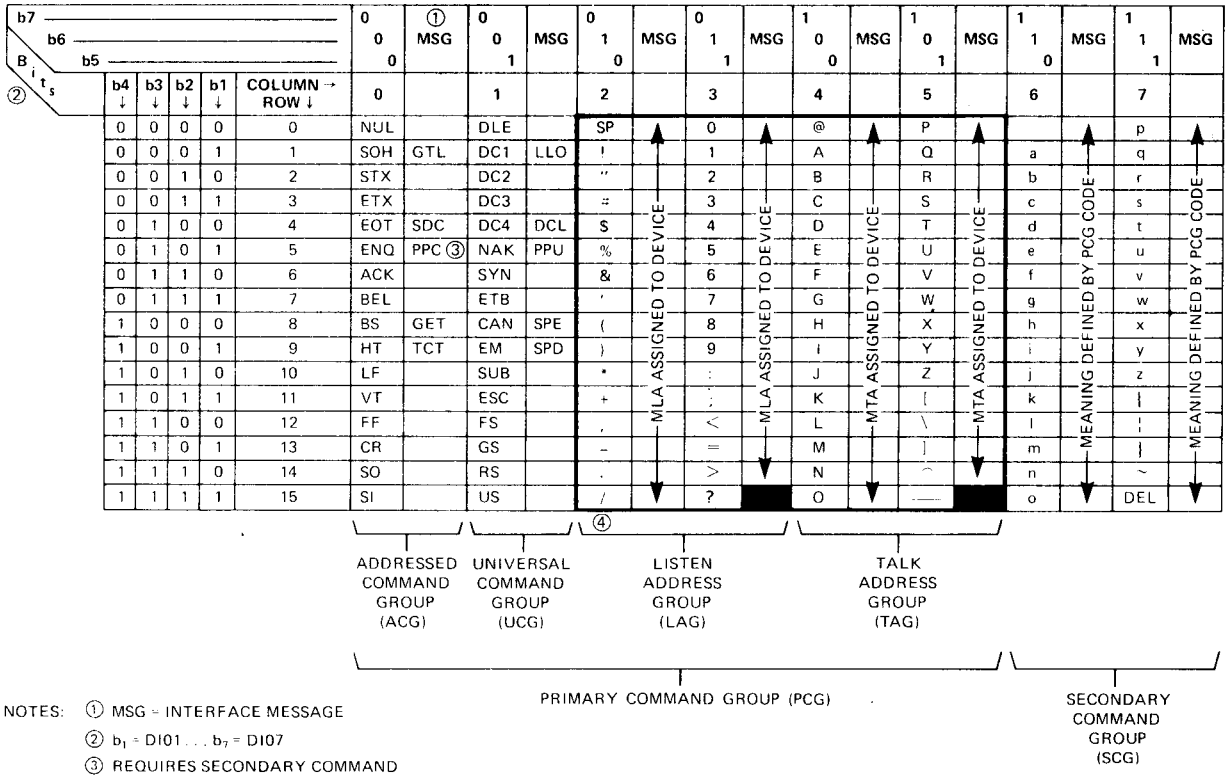
Table 7.24. ST status code for IEEE-488 bus.

IEEE-488 REGISTER ADDRESSES

Table 7-24 shows the IEEE-488 hardware addresses for the PET. An attempt to control the bus by means of the PEEK and POKE commands will fail, if the time out intervals for the 488 devices are exceeded.

Hex Address	Decimal Address	Bits	IEEE	Mode
E820	59424	0-7	DI01-8	Input
E822	59426	0-7	DI01-8	Output
E821	59425	3	NDAC	Output
E823	59427	3 7	DAV SRQ	Input
E810	59408	6	EOI	Input
E840	59456	0 1 2 6 7	NDAC NRFD ATN NRFD DAV	Input Output Output Input Output

Table 7.25. IEEE-488 hardware addresses and signal information.



NOTES:
① MSG = INTERFACE MESSAGE
② b_1 = DI01 . . . b_7 = DI07
③ REQUIRES SECONDARY COMMAND
④ DENSE SUBSET (COLUMN 2 THROUGH 5). ALL CHARACTERS USED IN BOTH COMMAND & DATA MODES.

Table 7.26. Code assignments for "Command Mode" of operation.
(SENT AND RECEIVED WITH ATN TRUE)

Chapter 8. USE OF THE PET FOR MACHINE LANGUAGE PROGRAMMING

Machine language programs execute much faster than do BASIC programs which have to be interpreted first then executed. On PET, machine language can be used to communicate with the user port, play music, or write the screen memory with blinding speed. If you have never programmed the 6502 microprocessor, it is probably adviseable that you get hold of the two books mentioned in Chapter 1 before you proceed with this chapter.

In PET there are two ways to create a machine language program in memory and execute it. The first is by BASIC. As previously discussed, there are two BASIC commands, PEEK and POKE which give equivalent machine language operation relative to controlling input/output instructions or influencing or sampling individual memory locations. The second method to program is by a monitor.

A monitor essentially has only three functions: examine and deposit bytes in memory, and branch to execute code. These functions are available as PEEK, POKE and SYS in BASIC. The chief limitation of BASIC is that all bytes must be converted to decimal before use. A monitor available for PET allows one to work entirely in hexadecimal notation but the 6502 does not care what base you work in because all it sees is binary. The PET monitor does have some other useful features which we will discuss later.

MACHINE LANGUAGE PROGRAMMING FROM BASIC

It is possible to build into a string of memory locations by means of a POKE command, a set of instructions which are a machine language subroutine which is usable by an individual program. To implement these subroutines, there are four basic considerations: (1) what the subroutine is supposed to do, (2) how to implement it, (3) where to put the program, and (4) how to communicate the subroutine from BASIC. The decision on what the program is to do and how to implement it is left to the programmer and the programming manual (6502).

To locate the code, you must decide whether you have a small program that is to be used only temporaril or whether it is a program you want to have operational throughput the entire time the BASIC program is operating in the machine.

To understand how best to keep the program in memory, we should review the memory map of the PET. All the zero page programs address are consumed by the operating system and are usually being changed throughout the programs. Between the normal use of stack and tape I/O corrections, all of page 1 is used. Page 2 has a series of variables which are again used throughout the program. However, memory locations 634 through 1023 are used for the first and second cassette buffers. If a program is not using tape I/O, then these areas will not be touched by BASIC.

If only the first cassette is used, the second cassette buffer is available. If both the cassettes are used during the program, or if this area is not enough into which the user is to write some code, then the space between the end of the BASIC program and where BASIC stores its variables is the space that is available to the programmer. At any time during execution of the program, a PEEK into location 124 and 125 indicates the beginning location of the BASIC variables. Working back down these with a small safety margin which is proportional to the amount of data space that is used in the program, is a memory area which is not affected by BASIC during execution. These are memory locations which are counted by the FRE statement. Once programs have been written and debugged, this space is as useful as are the cassette locations.

The final problem is how to get the program into the memory location. Although by use of the machine language monitor, machine language programs are loadable, this involves a two-step process for the user. First, the machine language program must be loaded, followed by the loading of the BASIC

program. Obviously, this technique does not work at all, if the program is to be loaded into the cassette buffers. Another technique is to assemble the program, into the BASIC program, by means of putting the machine language program into data statements. The data statements can then be read at the beginning of the execution of the BASIC program and POKEd into the appropriate memory locations.

SYS COMMAND

When it is necessary to transfer control to the machine language program, there are two ways to do it. The preferred approach is the SYS command which transfers control totally from BASIC until control is returned by means of a routine from subroutine instruction. It can be used to transfer control to any other program such as a machine language monitor or future languages when they become available. If the following code is encountered

10 SYS (634)

at Line 10, BASIC will hand control of the computer to the program located at 634. The general format for the SYS command is

SYS (start address)

The start address can be a computed value, in either case, it must result in a positive number not greater than 65535. NOTE: Execution of machine language code, removes almost all protection that the ROMs has built into it to allow the BASIC interpreter to continue functioning without regard to user error. *As soon as you transfer control from BASIC to your own program, any mistakes which occur in your program may cause the machine to cease to function.* In order to help solve this type of problem, you should use the machine language monitor to develop anything other than the most trivial amount of code. In any case, when control of system is lost, it can be regained by repowering the system on.

In order to return from the SYS command, the last instruction in the program, which is executed, should be a RTS instruction. BASIC will then start interpreting the next statement after the SYS command. In order to pass the variables of data back and forth between the user program and BASIC using the SYS command, data has to be POKEd into temporarily undisturbed memory locations during the execution of the BASIC routine. The results of the SYS operation would have to be PEEKed back into the program that follows the call to SYS.

USR FUNCTION

There are some programs, particularly mathematical ones, in which it is easier to pass parameters to/from BASIC using the USR function and to get the results directly processed in BASIC. USR is specified with a parameter. BASIC evaluates the expression for its parameter and leaves the results of the evaluation in a floating accumulator which BASIC uses for all of its functions. It is noted that if no parameter is passed, the floating accumulator is not initializeable by the user or by any other techniques as it is used by BASIC in a variety of ways prior to executing the USR function.

USR calls a routine, which executes a machine language program. A result in the floating accumulator to be analyzed by the BASIC expression. Because USR is a function, it is possible to include the function called user as part of a BASIC instruction as in: IF USR (A) = 1, THEN etc. In this case the parameter A will be passed to the USR function in the floating accumulator. The resulting floating accumulator, when the user returns to BASIC, would be compared to 1 and the logical function would be executed.

The SYS command is more useful for transferring control for machine language processing in which variables are not being acted on. USR is more useful when one is trying to implement a new BASIC command. This is an important consideration in using USR. USR uses preassigned variable locations: locations 1 and 2. These locations must be initialized with the hexadecimal value of the starting address in which the machine language program is stored. This can be done anywhere throughout the program

with a POKE of the decimal equivalent of the lower address to location 2 and POKE of the high order address in location 2. Example:

```
10POKE 1,122
20POKE 2,2
30 IF USR (A) = 1 THEN etc.
```

USEFUL BASIC SUBROUTINES

There are a series of subroutines in BASIC which can allow the machine language program to evaluate values in the floating accumulator. These functions are called jump to Subroutines instruction (JSR) to the address.

The parameter specified in the USR functon is evaluated, converted to a binary floating point equivalent with signs, exponent, and mantissa, and placed in a series of 6 bytes which we will call the floating accumulator

$B0	sign and exponent
$B1	mantissa MSB
$B2	mantissa
$B3	mantissa
$B4	mantissa
$B5	mantissa LSB
$B6	sign of mantissa

The exponent is computed such that the mantissa $0 = 1 \times 1$. It is stored as a signed 8 bit binary +$80. Negative exponents are not stored 2's complement. Maximum exponent is 10^{38}. Minimum exponent is 10^{-39} which is stored as $00. A zero exponent is used to flag the number as zero.

Exponent	Approximate Value
FF	10^{38}
A2	10^{10}
7F	10^{-1}
02	10^{-38}
00	10^{-39}

Since the exponent is really a power of 2, it should best be described as the number of left shifts (EXP>$80) or right shifts (EXP< = $80) to be performed on the normalized mantissa to create the actual binary representation of the value.

Since the mantissa is always normalized, the high order bit of the most significant byte is always set. This guarantees always at least 40 bits precision which is roughly equivalent to 9 significant digits plus a few bits for rounding. If a number has a value of zero, itmay not always have zero bytes in the mantissa. The only true flag for a zero number is the exponent. See Figure 8.1 for example exponents and mantissa's.

If the mantissa is positive, then the sign byte is zero -- $00. A negative mantissa causes this byte to be -1--$FF.

EXAMPLE FLOATING POINT NUMBERS

	exponent	mantissa	mantissa	mantissa	mantissa	Sign of mantissa
1E38	FF	96	76	99	52	00
4E10	A4	95	02	F9	00	00
2E10	A3	95	02	F9	00	00
1E10	A2	95	02	F9	00	00
1	81	80	00	00	00	00
.5	80	80	00	00	00	00
.25	7F	80	00	00	00	00
1E-4	73	D1	B7	59	59	00
1E-37	06	88	IC	14	14	00
1E-38	02	D9	C7	EE	EE	00
1E-39	00	A0	00	00	00	00
0	00	00	00	00	00	
−1	81	80	00	00	00	FF
−10	84	A0	00	00	00	FF

Figure 8.1.Example floating point numbers.

Actual floating point BASIC variables are stored in 5 bytes, rather than 6 bytes as is the floating accumulator. Upon examination, one will note that the most significant byte of the mantissa is always set. If we always assure the number will be in this format, we can use that bit to indicate the sign of the mantissa -- thus freeing the byte used for sign. The sixth byte is used in the floating accumulator to simplify operations when shifting the mantissa.

The contents of the floating accumulator may be converted to a double byte integer by calling a subroutine FLPINT which is located at $DOA7. The most significant byte of the integer is returned in $B3 and the least significant byte in $B4.

e.g

```
10 A = USR(2)
contents of FAC after USR call
82 80 00 00 00 00

JSR FLPINT
contents of FAC after conversion

82 00 00 [ 00 02 ] 00 00
         integer value
```

It is not necessary to return a value in the FAC after a USR call. The value of USR can be left as just the current contents of FAC. An integer can be converted back to floating by loading the most significant byte into index register Y then calling INTFLP at $D278.

e.g. L D A MSB
LDY LSB
JSR INTFLP

USEABLE I/O ROUTINES

Read a line, pass a character

$FFCF return char in 0
no other regs changed

Print a character on screen

$FFD2 Char in A
no regs changed

Test for stop key

$FFE1 returns =, <>
only A changed

Get a character from keyboard

$FFE4
char or if none then null (00)

SUMMARY

There are two ways to communicate from BASIC to machine language program. The simplest of these is SYS in which the control of the computer is turned over to the machine language program located at the address specified in thesys command. For implementing your own functions in BASIC, there is a function called USR which when memory locations of 1 and 2 are properly initialized to point in a machine language program, evaluate a parameter specified in the user function and pass the results back to the program using the floating accumulator. A series of useful subroutines, available in BASIC, can allow either the USR or SYS function to perform operations on the floating accumulator without the user running any program other than the calling routines.

In all cases, the use of the machine language program is only for the more sophisticated BASIC user. The protection of the ROM fail safe coding is lost. Machine language programs should only be used when BASIC is neither fast enough nor the function which is desired is implemented.

MACHINE LANGUAGE MONITOR

TIM is the Terminal Interface Monitor program for MOS Technology's 65XX microprocessors. It has been expanded and adapted to function on the Commodore PET. PET uses a cassette tape version of this monitor. Execution is transfered from the PET BASIC interpreter to TIM by the SYS command.

To LOAD your MONITOR, take the cassette with MONITOR and put it in the tape unit with the MONITOR side up. Then type: LOAD "MONITOR" and, when ready, RUN.

Commands typed on the PET keyboard can direct TIM to start executing a program, display or modify registers and memory locations, and load or save binary data. On modifying memory, TIM performs automatic read after write verification to insure that addressed memory exists, is R/W type, and is responding correctly.

TIM also provides several subroutines which may be called by user programs. These include reading and writing characters on the video display, typing a byte in hexadecimal and typing a CRLF sequence.

TIM COMMANDS

M	display memory
R	display register
G	begin execution
X	exit to BASIC
L	load
S	save

EXAMPLES

```
M      DISPLAY MEMORY
.M C000,C010
.: C000 1D C7 48 C6 35 CC EF C7
.: C008 C5 CA DF CA 70 CF 23 CB
.: C010 9C C8 9C C7 74 C7 1F C8
```

In a Display Memory command, the start and ending addresses must be completely specified as 4 digit hex numbers. To modify a memory location, move the cursor up in the display, type the correction and press RETURN to enter the change. When you move the cursor to a line to do a screen edit, and press RETURN, the colon tells the monitor that you are re-entering data.

```
R      DISPLAY REGISTERS
.R PC SR AC XR YR SP
.: C6 ED 00 20 00 F5
```

Registers are saved and restored upon each entry or exit from TIM. They may be modified or preloaded as in the display memory example above. The semicolon tells the monitor you are modifying registers.

```
G      BEGIN EXECUTION
.G C38B
```

The GO command may have an optional address for the target. If none is specified, the PC from the R command is taken as the target.

```
X      EXIT TO BASIC
.X
```

R E A D Y

Causes a warm start of BASIC. In a warm start memory is not altered in any way and BASIC resumes operation the way it was before a monitor was made.

```
L     LOAD
L 01 MONITOR
PRESS PLAY ON TAPE #1
OK
FOUND MONITOR
LOADING
```

No defaults are allowed on a LOAD command. The device number and the file name must be completely specified. Operating system prompts for operator intervention are the same as for BASIC. Memory addresses are loaded as specified in the file header which is set up by the SAVE command. Machine language subroutines may be loaded from BASIC but care must be taken not to use BASIC variables as the variable pointer is set to the last byte loaded + 1.

```
S      SAVE
.S 01,MONITOR,0400 ,076D

.PRESS PLAY ON TAPE#1
OK
```

Likewise, no defaults on the SAVE command. Any start and ending address may be specified.

To cancel a command either type RETURN or press STOP to cancel a Display Memory, LOAD or SAVE.

INTERRUPT AND BREAKPOINT ACTION

BRK is a software interrupt instruction which causes the CPU to interrupt execution, save PC and P registers on the stack and then branch through a vector at locations $021B and $021C. TIM initializes this vector to point at itself on entry by CALL. Unless the user modifies this vector, TIM will gain control when a BRK instruction is executed, print B* indicating entry via breakpoint (instead of C* entry via call) and the registers (as in the R command), and wait for user commands. Note that after a BRK which vectors to TIM, the user's PC points to the byte following the BRK: however, users who choose to handle BRK instructions themselves should note that BRK acts as a two-byte instruction, leaving the PC (on return via RTI)*two* bytes past the BRK instruction.

IRQ is vectored normally in PET to an ISR which updates the clock and scans the keyboard every 60*th* of a second. If the vector is altered and the machine language subroutine does not restore it, a power-on reset must be performed.

NMI is not provided for in the PET. The processor line corresponding to this interrupt is permanently pulled UP.

REST vectors to a cold-start of BASIC. Memory is cleared. Reload and re-enter TIM via SYS command.

TIM MONITORS CALLS AND SPECIAL LOCATIONS

JSR	WRT	$FFD2	type a character
JSR	RDT	$FFCF	input a character
JSR	GET	$FFE4	Get a character
JSR	CRLF	$04F2	type a CR
JSR	SPACE	$063A	type a space
JSR	WROB	$0613	type a byte
JSR	RDOB	$065E	read a byte
JSR	HEXIT	$0685	Ascii to hex in A

MEMORY USAGE

$0A-$22	zero page
$400-$76A	absolute RAM

$23-$5A are zero page locations in the BASIC input buffer which may be used when BASIC is not using these locations. The second cassette buffer **$33A-$3FF** is a well protected location if that device is not used. Other memory locations may be used with considerable risk, depending upon which piece of PET software wants to use it also.

MONITOR CHECKOUT PROCEDURE

1) Power up your PET normally into BASIC command mode. Insert the cassette containing a monitor and use the SHIFT-RUN sequence to initiate a program load. You should see a display something like:

```
C*  PC  SRAC  XR  YR  SP
.;  29  00 88 89  FE
```

Exact values may vary, although the first and last values should be as shown.

2) The display of registers is the standard entry display message. It consists of C* to identify entry by call, followed by the CPU register contents: program counter, processor status, accumulator, X index, Y index, and stack pointer. Note that all TIM inputs and outputs are in base 16 which is referred to as

hexadecimal, or just hex. In hexadecimal, the digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. After printing the CPU registers, TIM is ready to receive commands from you. TIM indicates this "ready" status by typing the prompting character "." on a new line.

3) The user's CPU register may also be displayed with the R command. Type an R and press RETURN. The monitor should respond as above, but without the asterisk.

4) Displayed values may be monitored by screen edit and re-entry of the line via return. Remember to type spaces to delimit fields and type 4 digit hex numbers for addresses and 2 digits for byte contents.

5) Memory may be displayed and modified using the M command. Type:

```
.M 0100 0107
```

You will see a display something like:

```
              0   1   3   4   5   6   7
.:  0100     20  00  30  30  30  30  30
```

Now use the screen edit to modify in place on the screen, type RETURN and display again.

6) Use M and ; to enter the following test program called CHSET because it prints the ASCII 64 character set on the terminal. The M command is used to display memory locations on the PET screen and it is then possible to use the screen edit on each line and type RETURN to alter memory.

```
* = $33A
CRLF = $4F2
WRT = $FFD2


033A  20  F2  04 ;  CHSET  JSR   CRLF
033D  A2  20               LDX   #$20
033F  8A           LOOP    TXA
340   20  D2  FF           JSR   WRT
343   E8                   INX
344   E0  60               CPX   #$60
346   D0  F7               BNE   LOOP
348   00                   BRK
349   4C  3A  03           JMP   CHSET

.M  033A,034B
.: 033A 20 F2 04 A2 20 8A 20 D2
.: 0342 FF E8 E0 60 D0 F7 00 4C
.: 034A 3A 03
```

7) CHSET was assembled to reside in the *2nd* cassette buffer. Type:

```
. G 033A
```

to execute the program.

The listing should look like this:

```
! " # $ % ' ( ) * , - . / 0 1 2 3 4 5 6 7 8 9 : ; = ? @ A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z [ / ]
B * PC  SR  AC  XR  YR  SP
.; 0349 3B 5F 60 8D FE
```

Note the address contained in the PC. It is possible to type G execute the program again without specifying an address.

8) Next we will link CHSET with BASIC. First replace the BRK instruction in location $348 with an RTS (return subroutine) (change $348 from 00 to 60).

9) Change the USR function vector in locations 1 and 2 to point at the subroutine $33A.

```
.: 0000 4C 3A 03
```

10) Exit from the monitor and re-enter BASIC.

```
.X
READY
```

11) Prove that the linkage is established by using both SYS and USR.

```
A = USR(0)
SYS (3*256 + 3*16 + 10)
```

(Enter these as direct commands.)

```
LINE # LOC     CODE       LINE

0002  0000              ;COPYRIGHT 1978 BY
0003  0000              ;COMMODORE INTERNATIONAL LIMITED

0005  0000              VARTAB=$7C
0006  0000              TXTPT=$CA
0007  0000              NCMDS=8
0008  0000              VPLI=$91
0009  0000              RDT=$FFCF
0010  0000              WRT=$FFD2
0011  0000              CBINV=$021B
0012  0000              WARM=$C38B
0013  0000              FA=$F1
0014  0000              FNLEN=$EE
0015  0000              FNADR=$F9
0016  0000              STAL=$F7
0017  0000              STAH=$F8
0018  0000              EAL=$E5
0019  0000              EAH=$E6
0020  0000              ;ZERO PAGE MONITOR RESERVE AREA
0021  0000              ;
0022  0000                    *=$0A
0023  000A              WRAP  *=*+1           ;ADDRESS WRAP-AROUND FLAG
0024  000B              DIFF  * =*+2
0025  000D              BRKF  *=*+1           ;BREAK FLAG
0026  000E              PREVC *=*+1
0027  000F              ACMD  *=*+2
0028  0011              TMP0  *=*+2
0029  0013              TMP2  *=*+2
0030  0015              TMP4  *=*+2
0031  0017              TMP6  *=*+2
0032  0019              PCL   *=*+1
0033  001A              PCH   *=*+1
0034  001B              FLGS  *=*+1
0035  001C              ACC   *=*+1
0036  001D              XR    *=*+1
0037  001E              YR    *=*+1
0038  001F              SP    *=*+1
0039  0020              SAVX  *=*+1
0040  0021              TMPC  *=*+1
0041  0022              TMPC2 *=*+1
0042  0023              RCNT=TMPC
0043  0023              LCNT=TMPC2
0044  0023              ISTR  *=*+16          ;FILEID BUFFER
0045  0033                    *=$400
0046  0400              ;ENTER COMPILED BASIC TEXT
0047  0400  0B                   .BYT 0,13,4,10,0,158
0047  0401  0D
0047  0402  04
0047  0403  0A
0047  0404  00
0047  0405  9E
0048  0406  28 31               .BYT '(1039)',0,0,0
0048  040C  00
0048  040D  00
0048  040E  00
```

```
LINE #  LOC     CODE        LINE

0049  040F                ;
0050  040F                ;CALL ENTRY POINT
0051  040F                ;       STACK CONTAINS Y,X,A,S,PC
0052  040F                ;BREAK ENTRY POINT
0053  040F                ;       STACK CONTAINS Y,X,A,S,PC
0054  040F                ;
0055  040F  A9 27          CALLE   LDA #<BRKE          ;INIT BRK VEC.
0056  0411  8D 1B 02              STA CBINV
0057  0414  A9 04                LDA #>BRKE
0058  0416  8D 1C 02              STA CBINV+1
0059  0419  A9 07                LDA #>EOM
0060  041B  85 7D                STA VARTAB+1
0061  041D  A9 6B                LDA #<EOM
0062  041F  85 7C                STA VARTAB
0063  0421  A9 43                LDA #'C           ;SET A=C TO INDICATE
0064  0423  85 21                STA TMPC
0065  0425  D0 12                BNE B3            ;CALL ENTRY, THEN JMP TO B3
0066  0427  A9 42          BRKE    LDA #'B         ;SET A=B FOR BREAK
0067  0429  85 21                STA TMPC
0068  042B  D8                   CLD
0069  042C  4A                   LSR A             ;SET CY FOR PC CORRECTION
0070  042D  68                   PLA
0071  042E  85 1E                STA YR            ;SAVE Y
0072  0430  68                   PLA
0073  0431  85 1D                STA XR            ;AND X
0074  0433  68                   PLA
0075  0434  85 1C                STA ACC           ;AND ACCUMULATOR
0076  0436  68                   PLA
0077  0437  85 1B                STA FLGS          ;AND FLAGS
0078  0439  68              B3      PLA
0079  043A  69 FF                ADC #$FF          ;CY SET TO PC-1 FOR BREAK
0080  043C  85 19                STA PCL
0081  043E  68                   PLA
0082  043F  69 FF                ADC #$FF
0083  0441  85 1A                STA PCH
0084  0443  BA                   TSX
0085  0444  86 1F                STX SP            ;SAVE ORIGINAL SP
0086  0446  58                   CLI               ;CLEAR INTS
0087  0447  20 F2 04        B5      JSR CRLF
0088  044A  A6 21                LDX TMPC          ;SET X EQUAL TO B OR C
0089  044C  A9 2A                LDA #'*
0090  044E  20 22 06              JSR WRTWO
0091  0451  A9 52                LDA #'R           ;SET FOR R DISPLAY TO
0092  0453                ;                       PERMIT IMMEDIATE
0093  0453                ;                       ALTER FOLLOWING
0094  0453                ;                       BREAKPOINT
0095  0453  85 0D                STA BRKF          ;SET BREAK FLAG
0096  0455  D0 2B                BNE S0
0097  0457  A9 00          START   LDA #0          ;NEXT COMMAND FROM USER
0098  0459  85 CA                STA TXTPT
0099  045B  85 0D                STA BRKF          ;CLEAR BREAK FLAG
0100  045D  85 0A                STA WRAP          ;CLEAR ADR WRAP-AROUND FLAG
0101  045F  20 F2 04              JSR CRLF
0102  0462  A9 2E                LDA #'.           ;TYPE A PROMPTING '.'
0103  0464  20 D2 FF              JSR WRT
```

```
LINE # LOC     CODE        LINE

0104  0467  A6 20              LDX SAVX        ;IF CURRENT CMD IS R OR M,
0105  0469                    ;                 SET CURSOR TO
0106  0469                    ;                 ALTER POSITION
0107  0469  E0 02              CPX #2
0108  046B  F0 04              BEQ ST0
0109  046D  E0 03              CPX #3
0110  046F  D0 06              BNE ST1
0111  0471  20 3A 06     ST0    JSR SPACE       ; POSITION UNDER PC DATA
0112  0474  20 37 06           JSR SPAC2
0113  0477  20 90 06     ST1    JSR RDUC        ;READ COMMAND. CHARACTER
0114  047A                    ;                 IS RETURNED IN A
0115  047A  C9 2E              CMP #'.         ;IGNORE PROMPTING '.'
0116  047C  F0 F9              BEQ ST1
0117  047E  C9 20              CMP #$20        ;IGNORE SPACES
0118  0480  F0 F5              BEQ ST1
0119  0482  A2 07        S0    LDX #NCMDS-1    ;LOOKUP COMMAND
0120  0484  DD 02 05     S1    CMP CMDS,X
0121  0487  D0 0F              BNE S2
0122  0489  A5 20              LDA SAVX        ;SAVE PREVIOUS COMMAND
0123  048B  85 0E              STA PREVC
0124  048D  86 20              STX SAVX        ;SAVE CURRENT COMMAND INDEX
0125  048F  BD 0A 05           LDA ADRH,X
0126  0492  48                 PHA
0127  0493  BD 12 05           LDA ADRL,X
0128  0496  48                 PHA
0129  0497  60                 RTS
0130  0498  CA           S2    DEX
0131  0499  10 E9              BPL S1          ;LOOP FOR ALL COMMANDS
0132  049B  A9 3F        ERRDPR LDA #$3F        ;OPERATOR ERROR RESTART
0133  049D  20 D2 FF           JSR WRT
0134  04A0  4C 57 04           JMP START       ;JMP START (WRT RETURNS  CY=
0135  04A3  38           DCMP   SEC             ;TMP2-TMP0 DOUBLE SUBTRACT
0136  04A4  A5 13              LDA TMP2
0137  04A6  E5 11              SBC TMP0
0138  04A8  85 0B              STA DIFF
0139  04AA  A5 14              LDA TMP2+1
0140  04AC  E5 12              SBC TMP0+1
0141  04AE  A8                 TAY             ;RETURN HIGH ORDER PART IN Y
0142  04AF  05 0B              ORA DIFF        ;OR LO FOR EQU TEST
0143  04B1  60                 RTS
0144  04B2  A5 11        PUTP   LDA TMP0        ;MOVE TMP0 TO PCH,PCL
0145  04B4  85 19              STA PCL
0146  04B6  A5 12              LDA TMP0+1
0147  04B8  85 1A              STA PCH
0148  04BA  60                 RTS
0149  04BB                    ;
0150  04BB                    ;DISPLAY MEM SUBR. SET AR=NUMBER
0151  04BB                    ;OF MEMORY BYTES DISPLAYED.
0152  04BB                    ;TMP0=ADR OF MEM DISPLAYED
0153  04BB                    ;
0154  04BB  85 21        DM     STA TMPC
0155  04BD  A0 00              LDY #0
0156  04BF  20 3A 06     DM1    JSR SPACE       ;WR N BYTES
0157  04C2  B1 11              LDA (TMP0),Y    ;(TMP0)=ADR
0158  04C4  20 13 06           JSR WR0B
```

```
LINE # LOC     CODE        LINE

0159  04C7  20 F7 04           JSR INCTMP
0160  04CA  C6 21              DEC TMPC
0161  04CC  D0 F1              BNE DM1
0162  04CE  60                 RTS
0163  04CF                    ;
0164  04CF                    ;READ AND STORE BYTE.
0165  04CF                    ;NO STORE IF SPACE OR RCNT = 0.
0166  04CF                    ;
0167  04CF  20 5E 06     BYTE  JSR RDDB          ;CHAR IN A, CY=0 IF SP
0168  04D2  90 0D              BCC BY3           ;SPACE
0169  04D4  A2 00              LDX #0            ;STORE BYTE
0170  04D6  81 11              STA (TMP0,X)
0171  04D8  C1 11              CMP (TMP0,X)      ;TEST FOR VALID WRITE (RAM)
0172  04DA  F0 05              BEQ BY3
0173  04DC  68                 PLA               ;ERROR: CLEAR JSR ADR IN STA
0174  04DD  68                 PLA
0175  04DE  4C 9B 04           JMP ERROPR
0176  04E1  20 F7 04     BY3   JSR INCTMP        ;GO INC TMP0 ADR
0177  04E4  C6 21              DEC RCNT
0178  04E6  60                 RTS
0179  04E7  A9 1B        SETR  LDA #FLGS         ;SET TO ACCESS REGS
0180  04E9  85 11              STA TMP0
0181  04EB  A9 00              LDA #0
0182  04ED  85 12              STA TMP0+1
0183  04EF  A9 05              LDA #5
0184  04F1  60                 RTS
0185  04F2  A9 0D        CRLF  LDA #$D
0186  04F4  4C D2 FF           JMP WRT
0187  04F7                    ;
0188  04F7                    ;INCREMENT (TMP0,TMP0+1) BY 1
0189  04F7                    ;
0190  04F7  E6 11        INCTMP INC TMP0         ;LOW BYTE
0191  04F9  D0 06              BNE SETWR
0192  04FB  E6 12              INC TMP0+1        ;HIGH BYTE
0193  04FD  D0 02              BNE SETWR
0194  04FF  E6 0A              INC WRAP          ;POINTER HAS WRAPPED
0195  0501                    ;                   AROUND: SET FLAG
0196  0501  60           SETWR  RTS
```

```
LINE # LOC      CODE        LINE

0198  0502  3A            CMDS    .BYTE ':'
0199  0503  3B                    .BYTE ';'
0200  0504  52                    .BYTE 'R'
0201  0505  4D                    .BYTE 'M'
0202  0506  47                    .BYTE 'G'
0203  0507  58                    .BYTE 'X'
0204  0508  4C                    .BYTE 'L'
0205  0509  53                    .BYTE 'S'
0206  050A  05            ADRH    .BYT >ZZ1
0207  050B  05                    .BYT >ZZ2
0208  050C  05                    .BYT >ZZ3
0209  050D  05                    .BYT >ZZ4
0210  050E  05                    .BYT >ZZ5
0211  050F  05                    .BYT >ZZ6
0212  0510  06                    .BYT >ZZ7
0213  0511  06                    .BYT >ZZ8
0214  0512  C1            ADRL    .BYT <ZZ1
0215  0513  B1                    .BYT <ZZ2
0216  0514  2C                    .BYT <ZZ3
0217  0515  5E                    .BYT <ZZ4
0218  0516  D7                    .BYT <ZZ5
0219  0517  FD                    .BYT <ZZ6
0220  0518  9E                    .BYT <ZZ7
0221  0519  9E                    .BYT <ZZ8
```

```
LINE # LOC     CODE       LINE

0223  051A  20 50        REGK    .BYTE ' PC  SR AC XR YR SP'
0224  052D              ;
0225  052D  A5 0D        DSPLYR  LDA BRKF          ;IF NOT BREAK ENTRY,
0226  052F  D0 06                BNE D1            ;CRLF,SPACE2
0227  0531  20 F2 04             JSR CRLF
0228  0534  20 37 06             JSR SPAC2
0229  0537  20 37 06     D1      JSR SPAC2
0230  053A  A2 00                LDX #0
0231  053C  BD 1A 05     D2      LDA REGK,X
0232  053F  20 D2 FF             JSR WRT
0233  0542  E8                   INX
0234  0543  E0 13                CPX #19
0235  0545  D0 F5                BNE D2
0236  0547  20 F2 04             JSR CRLF
0237  054A  A2 2E                LDX #'.
0238  054C  A9 3B                LDA #';
0239  054E  20 22 06             JSR WRTWO
0240  0551  20 37 06             JSR SPAC2
0241  0554  20 88 06             JSR WRPC          ;WRITE PC
0242  0557  20 E7 04             JSR SETR
0243  055A  20 BB 04             JSR DM            ;USE DM SUBR.
0244  055D  F0 4D                BEQ BEQS1
0245  055F  20 90 06     DSPLYM  JSR RDOC
0246  0562  20 4F 06             JSR RDOA          ;READ START ADR
0247  0565  90 48                BCC ERRS1         ;ERR IF NO SA
0248  0567  20 3F 06             JSR T2T2          ;SA TO TMP2
0249  056A  20 90 06             JSR RDOC          ;SKIP DELIMITER
0250  056D  20 4F 06             JSR RDOA          ;READ END ADR
0251  0570  90 3D                BCC ERRS1         ;ERR IF NO EA
0252  0572  20 3F 06             JSR T2T2          ;SA TO TMP0, EA TO TMP2
0253  0575  A0 00                LDY #0
0254  0577  B9 4A 07     COLH    LDA HDR,Y
0255  057A  30 06                BMI COLH2
0256  057C  20 D2 FF             JSR WRT
0257  057F  C8                   INY
0258  0580  D0 F5                BNE COLH
0259  0582  29 7F        COLH2   AND #$7F
0260  0584  20 D2 FF             JSR WRT
0261  0587  20 2A F3     DSP1    JSR TSTP          ;TEST FOR STOP KEY
0262  058A  F0 20                BEQ BEQS1
0263  058C  A6 0A                LDX WRAP          ;IF ADRS WRAP-AROUND, STOP
0264  058E  D0 1C                BNE BEQS1
0265  0590  20 A3 04             JSR DCMP
0266  0593  90 17                BCC BEQS1         ;STOP IF EA LESS THAN SA
0267  0595  20 F2 04             JSR CRLF          ;BEGIN NEW OUTPUT LINE
0268  0598  A2 2E                LDX #'.
0269  059A  A9 3A                LDA #':
0270  059C  20 22 06             JSR WRTWO
0271  059F  20 37 06             JSR SPAC2
0272  05A2  20 04 06             JSR WROA
0273  05A5  A9 08                LDA #8
0274  05A7  20 BB 04             JSR DM            ;DISPLAY 8, INCR TMP0
0275  05AA  F0 DB                BEQ DSP1
0276  05AC  4C 57 04     BEQS1   JMP START
0277  05AF  4C 9B 04     ERRS1   JMP ERROPR
```

```
LINE # LOC     CODE        LINE

0278  05B2                ;
0279  05B2                ;ALTER REGISTERS
0280  05B2                ;
0281  05B2  20 5E 06     ALTR   JSR RDOB          ;SKIP 2 SPACES
0282  05B5  20 4F 06            JSR RDOA          ;CY=0 IF SP
0283  05B8  90 03               BCC AL2           ;SPACE
0284  05BA  20 B2 04            JSR PUTP          ;ALTER PC
0285  05BD  20 E7 04     AL2    JSR SETR          ;SET TO ALTER R'S
0286  05C0  D0 0A               BNE A4
0287  05C2                ;
0288  05C2                ;ALTER MEMORY - READ ADR AND DATA
0289  05C2                ;
0290  05C2  20 5E 06     ALTM   JSR RDOB          ;SKIP 2 SPACES
0291  05C5  20 4F 06            JSR RDOA          ;READ MEM ALTER ADR
0292  05C8  90 E5               BCC ERRS1         ;CY=0, IF SPACE, ERR
0293  05CA  A9 08               LDA #8            ;SET CNT = 8
0294  05CC  85 21        A4     STA RCNT
0295  05CE  20 90 06     A5     JSR RDOC
0296  05D1  20 CF 04            JSR BYTE
0297  05D4  D0 F8               BNE A5
0298  05D6  F0 D4        A9     BEQ BEQS1
0299  05D8  20 CF FF     GO     JSR RDT
0300  05DB  C9 0D               CMP #$0D          ;IF CR, EXIT
0301  05DD  F0 0C               BEQ G1
0302  05DF  C9 20               CMP #$20          ;IF NOT SPACE, ERR
0303  05E1  D0 CC               BNE ERRS1
0304  05E3  20 4F 06            JSR RDOA
0305  05E6  90 03               BCC G1
0306  05E8  20 B2 04            JSR PUTP
0307  05EB  A6 1F        G1     LDX SP
0308  05ED  9A                  TXS               ;ORIG OR NEW SP VALUE TO SP
0309  05EE  A5 1A               LDA PCH
0310  05F0  48                  PHA
0311  05F1  A5 19               LDA PCL
0312  05F3  48                  PHA
0313  05F4  A5 1B               LDA FLGS
0314  05F6  48                  PHA
0315  05F7  A5 1C               LDA ACC
0316  05F9  A6 1D               LDX XR
0317  05FB  A4 1E               LDY YR
0318  05FD  40                  RTI
0319  05FE  A6 1F        EXIT   LDX SP
0320  0600  9A                  TXS
0321  0601  4C 8B C3            JMP WARM          ;EXIT TO BASIC WARM START
0322  0604                ;
0323  0604                ;WRITE ADR FROM TMP0 STORES
0324  0604                ;
0325  0604  A2 01        WROA   LDX #1
0326  0606  D0 02               BNE WROA1
0327  0608  A2 09        WRPC   LDX #9
0328  060A  B5 10        WROA1  LDA TMP0-1,X
0329  060C  48                  PHA
0330  060D  B5 11               LDA TMP0,X
0331  060F  20 13 06            JSR WROB
0332  0612  68                  PLA
```

```
LINE #  LOC     CODE        LINE

0333  0613                 ;
0334  0613                 ;WRITE BYTE --- A = BYTE
0335  0613                 ;UNPACK BYTE DATA INTO TWO ASCII
0336  0613                 ;CHARACTERS.  A=BYTE; X,A=CHARS
0337  0613                 ;
0338  0613  48            WROB    PHA
0339  0614  4A                    LSR A
0340  0615  4A                    LSR A
0341  0616  4A                    LSR A
0342  0617  4A                    LSR A
0343  0618  20 2B 06              JSR ASCII         ;CONVERT TO ASCII
0344  061B  AA                    TAX
0345  061C  68                    PLA
0346  061D  29 0F                 AND #$0F
0347  061F  20 2B 06              JSR ASCII
0348  0622                 ;
0349  0622                 ;WRITE 2 CHARS--X,A=CHARS
0350  0622                 ;
0351  0622  48            WRTWO   PHA
0352  0623  8A                    TXA
0353  0624  20 D2 FF              JSR WRT
0354  0627  68                    PLA
0355  0628  4C D2 FF              JMP WRT
0356  062B  18            ASCII   CLC
0357  062C  69 06                 ADC #6
0358  062E  69 F0                 ADC #$F0
0359  0630  90 02                 BCC ASC1
0360  0632  69 06                 ADC #$06
0361  0634  69 3A          ASC1    ADC #$3A
0362  0636  60                    RTS
0363  0637  20 3A 06       SPAC2   JSR SPACE
0364  063A  A9 20          SPACE   LDA #$20
0365  063C  4C D2 FF              JMP WRT           ;TYPE SP
0366  063F  A2 02          T2T2    LDX #2
0367  0641  B5 10          T2T21   LDA TMP0-1,X
0368  0643  48                    PHA
0369  0644  B5 12                 LDA TMP2-1,X
0370  0646  95 10                 STA TMP0-1,X
0371  0648  68                    PLA
0372  0649  95 12                 STA TMP2-1,X
0373  064B  CA                    DEX
0374  064C  D0 F3                 BNE T2T21
0375  064E  60                    RTS
0376  064F                 ;
0377  064F                 ;READ HEX ADR, RETURN HI IN TMP0,
0378  064F                 ;LO IN TMP0+1, AND CY=1
0379  064F                 ;IF SP CY=0
0380  064F                 ;
0381  064F  20 5E 06       RDOA    JSR RDOB          ;READ 2-CHAR BYTE
0382  0652  90 02                 BCC RDOA2         ;SPACE
0383  0654  85 12                 STA TMP0+1
0384  0656  20 5E 06       RDOA2   JSR RDOB
0385  0659  90 02                 BCC RDEXIT        ;SP
0386  065B  85 11                 STA TMP0
0387  065D  60            RDEXIT  RTS
```

```
LINE # LOC     CODE        LINE

0388  065E                ;
0389  065E                ; READ HEX BYTE AND RETURN IN A
0390  065E                ; AND CY=1 IF SP CY=0
0391  065E                ;
0392  065E  A9 00         RDOB    LDA #0          ;SPACE
0393  0660  85 0F                 STA ACMD        ;READ NEXT CHAR
0394  0662  20 90 06              JSR RDOC
0395  0665  C9 20         RDOB1   CMP #'
0396  0667  D0 09                 BNE RDOB2
0397  0669  20 90 06              JSR RDOC        ;READ NEXT CHAR
0398  066C  C9 20                 CMP #'
0399  066E  D0 0E                 BNE RDOB3
0400  0670  18                    CLC             ;CY=0
0401  0671  60                    RTS
0402  0672  20 85 06     RDOB2   JSR HEXIT       ;TO HEX
0403  0675  0A                    ASL A
0404  0676  0A                    ASL A
0405  0677  0A                    ASL A
0406  0678  0A                    ASL A
0407  0679  85 0F                 STA ACMD
0408  067B  20 90 06              JSR RDOC        ;2ND CHAR ASSUMED HEX
0409  067E  20 85 06     RDOB3   JSR HEXIT
0410  0681  05 0F                 ORA ACMD
0411  0683  38                    SEC             ;CY=1
0412  0684  60                    RTS
0413  0685  C9 3A         HEXIT   CMP #$3A
0414  0687  08                    PHP             ;SAVE FLAGS
0415  0688  29 0F                 AND #$0F
0416  068A  28                    PLP
0417  068B  90 02                 BCC HEX09       ;0-9
0418  068D  69 08                 ADC #8          ;ALPHA ADD 8+CY=9
0419  068F  60            HEX09   RTS
0420  0690  20 CF FF     RDOC    JSR RDT         ;READ CHAR
0421  0693  C9 0D                 CMP #$0D        ;IS IT A CR
0422  0695  D0 F8                 BNE HEX09       ;NO, RTS
0423  0697  68                    PLA
0424  0698  68                    PLA             ;YES, CLEAN STACK, EXIT
0425  0699  4C 57 04              JMP START
0426  069C  4C 9B 04     ERRL    JMP ERROPR
0427  069F  20 90 06     LD      JSR RDOC
0428  06A2  A9 00                 LDA #0
0429  06A4  85 EE                 STA FNLEN
0430  06A6  85 FA                 STA FNADR+1
0431  06A8  A9 23                 LDA #<ISTR
0432  06AA  85 F9                 STA FNADR
0433  06AC  20 5E 06              JSR RDOB READ FA
0434  06AF  29 0F                 AND #$F
0435  0681  85 F1                 STA FA          ;FIRST ADR
0436  06B3  20 90 06              JSR RDOC        ;SKIP COMMA
0437  06B6  A2 00                 LDX #0
0438  06B8  20 CF FF     RD2     JSR RDT
0439  06BB  C9 2C                 CMP #',
0440  06BD  F0 55                 BEQ L4
0441  06BF  C9 0D                 CMP #$D
0442  06C1  F0 0B                 BEQ L3
```

```
LINE # LOC     CODE        LINE

0443  06C3  E0 10           CPX #16
0444  06C5  F0 F1           BEQ RD2
0445  06C7  95 23           STA ISTR,X
0446  06C9  E6 EE           INC FNLEN
0447  06CB  E8              INX
0448  06CC  D0 EA           BNE RD2
0449  06CE  A5 20     L3    LDA SAVX        ;IS THIS A LOAD
0450  06D0  C9 06           CMP #6
0451  06D2  D0 C8           BNE ERRL        ;NO. ERROR
0452  06D4  A2 00     LD2   LDX #0
0453  06D6  8E 00 02        STX VERCK
0454  06D9  A5 F1           LDA FA
0455  06DB  D0 03           BNE *+5
0456  06DD  4C 9B 04  LD10  JMP ERROPR
0457  06E0  C9 03           CMP #3
0458  06E2  B0 F9           BCS LD10
0459  06E4  20 67 F6        JSR ZZZ
0460  06E7  20 3B F8        JSR CSTE1
0461  06EA  20 FF F3        JSR LD300
0462  06ED  A5 EE           LDA FNLEN
0463  06EF  F0 08           BEQ LD150
0464  06F1  20 95 F4        JSR FAF
0465  06F4  D0 00           BNE LD170
0466  06F6  4C 9B 04  LD120 JMP ERROPR
0467  06F9  20 AE F5  LD150 JSR FAH
0468  06FC  F0 F8           BEQ LD120
0469  06FE  20 4D F6  LD170 JSR LDAD2
0470  0701  20 22 F4        JSR LD400
0471  0704  20 8A F8        JSR TRD
0472  0707  20 13 F9        JSR TWAIT
0473  070A  AD 0C 02        LDA SATUS
0474  070D  29 10           AND #SPERR
0475  070F  D0 E5           BNE LD120
0476  0711  4C 57 04        JMP START
0477  0714  20 4F 06  L4    JSR RDDA        ;RD, STORE BEGIN'G ADR
0478  0717  A5 11           LDA TMP0
0479  0719  85 F7           STA STAL
0480  071B  A5 12           LDA TMP0+1
0481  071D  85 F8           STA STAH
0482  071F  20 CF FF  L5    JSR RDT         ;RD NXT CHAR
0483  0722  C9 20           CMP #$20
0484  0724  F0 F9           BEQ L5          ;IGNORE BLANKS
0485  0726  C9 0D           CMP #$D
0486  0728  F0 A4           BEQ L3          ;CR. GO TEST IF LOAD
0487  072A  C9 2C           CMP #','
0488  072C  F0 03           BEQ *+5
0489  072E  4C 9C 06        JMP ERRL
0490  0731  20 4F 06        JSR RDDA        ;RD, STORE ENDING ADRS
0491  0734  A5 11           LDA TMP0
0492  0736  85 E5           STA EAL
0493  0738  A5 12           LDA TMP0+1
0494  073A  85 E6           STA EAH
0495  073C  A5 20           LDA SAVX        ;TEST IF LOAD OR SAVE
0496  073E  C9 06           CMP #6
0497  0740  F0 92           BEQ LD2         ;GO LOAD
```

```
LINE # LOC     CODE        LINE

0498  0742  A2 00                LDX #0
0499  0744  20 B1 F6 63159       JSR SAVE          ;SAVE
0500  0747  4C 57 04             JMP START
0501  074A  0D           HDR     .BYT $0D,'
0501  074B  20 20
0502  0755  30 20                .BYT '0  1  2  3  4  5  6  ',$B7
0502  076A  B7
0503  076B                      TSTP=$F32A
0504  076B                      SV=LD
0505  076B                      ZZZ=$F667
0506  076B                      CSTE1=$F83B
0507  076B                      LD300=$F3FF
0508  076B                      SPERR=16
0509  076B                      SATUS=$20C
0510  076B                      TWAIT=$F913
0511  076B                      TRD=$F88A
0512  076B                      LD400=$F422
0513  076B                      LDAD2=$F64D
0514  076B                      FAH=$F5AE
0515  076B                      SAVE=$F681
0516  076B                      VERCK=$28B
0517  076B                      FAF=$F495
0518  076B                      ZZ1=ALTM-1
0519  076B                      ZZ2=ALTR-1
0520  076B                      ZZ3=DSPLYR-1
0521  076B                      ZZ4=DSPLYM-1
0522  076B                      ZZ5=GO-1
0523  076B                      ZZ6=EXIT-1
0524  076B                      ZZ7=LD-1
0525  076B                      ZZ8=SV-1
0526  076B                      EOM     END

ERRORS = 0000
```

```
SYMBOL TABLE

SYMBOL    VALUE

 A4        05CC     A5        05CE     A9        05D6     ACC       001C
 ACMD      000F     ADRH      050A     ADRL      0512     AL2       05BD
 ALTM      05C2     ALTR      05B2     ASC1      0634     ASCII     062B
 B3        0439     B5        0447     BEQS1     05AC     BRKE      0427
 BRKF      000D     BY3       04E1     BYTE      04CF     CALLE     040F
 CBINV     021B     CMDS      0502     COLH      0577     COLH2     0582
 CRLF      04F2     CSTE1     F83B     D1        0537     D2        053C
 DCMP      04A3     DIFF      000B     DM        04BB     DM1       04BF
 DSP1      0587     DSPLYM    055F     DSPLYR    052D     EAH       00E6
 EAL       00E5     EOM       076B     ERRL      069C     ERROPR    049B
 ERRS1     05AF     EXIT      05FE     FA        00F1     FAF       F495
 FAH       F5AE     FLGS      001B     FNADR     00F9     FNLEN     00EE
 G1        05EB     GO        05D8     HDR       074A     HEX09     068F
 HEXIT     0685     INCTMP    04F7     ISTR      0023     L3        06CE
 L4        0714     L5        071F     LCNT      0022     LD        069F
 LD10      06DD     LD120     06F6     LD150     06F9     LD170     06FE
 LD2       06D4     LD300     F3FF     LD400     F422     LDAD2     F64D
 NCMDS     0008     PCH       001A     PCL       0019     PREVC     000E
 PUTP      04B2     RCNT      0021     RD2       06B8     RDEXIT    065D
 RDOA      064F     RDOA2     0656     RDOB      065E     RDOB1     0665
 RDOB2     0672     RDOB3     067E     RDOC      0690     RDT       FFCF
 REGK      051A     S0        0482     S1        0484     S2        0498
 SATUS     020C     SAVE      F6B1     SAVX      0020     SETR      04E7
 SETWR     0501     SP        001F     SPAC2     0637     SPACE     063A
 SPERR     0010     ST0       0471     ST1       0477     STAH      00F8
 STAL      00F7     START     0457     SV        069F     T2T2      063F
 T2T21     0641     TMP0      0011     TMP2      0013     TMP4      0015
 TMP6      0017     TMPC      0021     TMPC2     0022     TRD       F88A
 TSTP      F32A     TWAIT     F913     TXTPT     00CA     UPL1      0091
 VARTAB    007C     VERCK     020B     WARM      C38B     WRAP      000A
 WROA      0604     WROA1     060A     WROB      0613     WRPC      0608
 WRT       FFD2     WRTWO     0622     XR        001D     YR        001E
 ZZ1       05C1     ZZ2       05B1     ZZ3       052C     ZZ4       055E
 ZZ5       05D7     ZZ6       05FD     ZZ7       069E     ZZ8       069E
 ZZZ       F667
END OF ASSEMBLY
```

Chapter 9. ERRORS AND DIAGNOSTICS

One of the advantages of the highly interactive way in which you are able to use your PET is that errors are easily correctable, due to the fact that the languages that are used within the machine have specific rules under which the not so smart computer can operate. These rules are necessary to allow the language to be able to understand what you are trying to tell it. Whenever BASIC cannot perform a function, it will tell you about it in the form of an error message. A total list of the error messages and some examples of what causes them follows.

The advantage of having this immediate response on the screen is that you can use the screen editor to immediately fix the problem as it occurs. In most cases, the problem is going to be obvious to you. The most common error is the syntax error problem, which means that you have typed the line to BASIC that it doesn't understand. The correction for this type of problem is to list the line that is being complained about and compare the typed data to what you thought you were going to type. About 90 percent of the time, you will discover the mistake by superficial inspection. If not, you may have to make reference to the appendix which defines the form for all the BASIC statements and if that does not clarify it for you, go to the individual write-up to understand what you are doing wrong.

The common problems are you have got a comma in the wrong place, or you used a variable that cannot be used in this particular kind of format. The basic premise to remember when correcting errors is that although the language is forgiving of exact requirements for spaces verses no spaces etc., that the rules are explicit. If you violate the rules, the computer is going to continue to complain about an error until you give it a problem it understands. Sometimes, the error is not as easy to undestand, although in almost all cases while executing a problem, if an error is encountered, the line number will be indicated.

Sometimes a problem is the result of a programming mistake that you have made in a previous computation. For instance, if you get a divide by zero in line 75 and you know you shouldn't be dividing by zero because, in your opinion, the value that is in the divisor should never be zero. The error is probably not on line 75, but somewhere further up your program where you define the variable. In order to attack this kind of problem, the use of temporary print statements is the common technique. In other words, if the variable is zero on line 75 and you don't think it should be, then you should list the portion that defines the variable. More often than not, an inspection of this area will show the problem to you immediately. If not, insert lines at appropriate places where the variable is computed to see when the variable acquires a value that you don't expect. This technique will usually allow you to figure out the problem in your programming.

The error messages in PET BASIC have been expanded over those other BASICs to give you a readily English format for what the message is. However, other than using the techniques which we have just described, the computer cannot fix a problem for you, it is in this area that programmers are made or broken. Just remember that nobody is looking over your shoulder and use the machine to help you understand the problem. If necessary, write little test routines which do only a piece of your program, until you understand what is causing your problem.

ERROR MESSAGES

On encountering an error in interpretation of a statement, whether in direct or program execution, BASIC displays a diagnostic message then returns to direct mode.

```
?MESSAGE ERROR IN LINE NUMBER
READY.
```

Resumption of execution is not permitted with a CONT command. Variables within the statement or program retain their values so they may be scrutinized to determine a cause of error, if necessary. GOSUB and FOR entries on the stack at the time of error are cleared so resumption of execution is not possible by RETURN or NEXT.

POSSIBLE BASIC MESSAGES AND MEANINGS

Bad subscript- An attempt was made to reference a matrix element which is outside the dimensions of the matrix. This may happen by specifying the wrong number of dimensions or a subscript larger than specified in the original dimension.

```
DIM A(2,2)
A(1,1,1)=2
?BAD SUBSCRIPT ERROR
READY.
A(10,10)=2
?BAD SUBSCRIPT ERROR
READY.
```

Can't continue-Program execution cannot be resumed via a CONT command in four cases:

1) no program exists.
2) a new line was just typed in.
3) the program has not recently been run.
4) an error just ocurred.

```
10A$='HELLO'
CONT
'CAN'T CONTINUE ERROR'
READY.
```

Division by zero-Zero as a divisor would result in numeric overflow--thus it is not allowed. When this message appears, it is most expedient to list the statement and look for division operators.

```
?DIVISION BY ZERO ERROR IN 10
LIST 10
10A=B/C
?C
0
```

Formula too complex--This message concerns only string expressions when BASIC runs out of string temporary pointers to keep track of substrings in evaluating a string expression.

```
?FORMULA TOO COMPLEX ERROR
READY.
```

Break the string expression into two smaller parts to cure the problem.

Illegal direct--A single 80 column buffer area is used by BASIC to process incoming characters. This same buffer is used to hold a statement that is being interpreted in direct mode. INPUT will not work because incoming characters would overwrite the variable list following INPUT to be processed.

DEF cannot be used in direct mode for a different but similar reason. The name of a function is stored in the BASIC variable area with pointers to the string of caharacters which define the function. Since the function exists only in the input buffer, it would be wiped out the first time a new command is typed-in.

```
INPUT A
?ILLEGAL DIRECT ERROR
READY.
```

Illegal quantity--Occurs when a function is accessed with a parameter out of range. This error may be caused by:

1. A matrix subscript out of range 0< X ≤ 32767
```
X(-1)=Y
?ILLEGAL QUANTITY ERROR
```
2. LOG (negative or zero argument)
3. SQR (negative argument)
4. A ↑ B where A=0 and B not integer.
?(-5)^ is illegal because it would give a complex result.
5. Call of USR before machine language subroutine has been patched in.
6. Use of string functions MID$, LEFT$, RIGHT$, with length parameters out of range(1< X < 255).
7. Index on ....GOTO out of range.
8. addresses specified for PEEK, POKE, WAIT and SYS out of range. (0 < X <65535).
9. Byte parameters of WAIT, POKE, TAB and SPC out of range (0< X<255).

```
POKE 32768,1000
?ILLEGAL QUANTITY ERROR
READY.
```

Next without for--Either a NEXT is improperly nested or the variable in a NEXT statement corresponds to no previously executed FOR statement.

```
FOR I=1 TO 10:NEXT:NEXT
?NEXT WITHOUT FOR ERROR
READY.

FOR I=1 TO 10:NEXT J
?NEXT WITHOUT FOR ERROR
READY.
```

OUT OF DATA--A READ statement was executed but all of the data statements in the program have been read. The program tried to read too much data, or insufficient

OUT OF DATA--A READ statement was executed but all of the data statements in the program have been nt data, was included in the program. Carriage

OUT OF DATA--A READ statement was executed but all of the data statements in the program have been read. The program tried to read too much data, or insufficient data, was included in the program. Carriage returning through a line READY on the PET TV display, sometimes yields this error because the message is interpreted as READ Y.

```
READY.
?OUT OF DATA ERROR
READY.
```

OUT OF MEMORY--May appear while entering or editing a program as the text completely fills memory. At run time, assignment and creation of variables may also fill all variable memory. Array available declarations consume large areas of memory even though a program may be rather short. The maximum number of FOR loops and simultaneous GOSUBs are dependent on each other. This context is stored on the 6502 hardware stack whose capacity may be exceeded. To determine the type of memory error, print FRE (0). If there are a large number of bytes variables, it is most likely a FOR-NEXT or GOSUB problem.

```
10GOSUB10
RUN
?OUT OF MEMORY ERROR IN 10
READY.
?FRE(0)
7156
```

OVERFLOW--Numbers resulting from computations or input that are larger than 1.70141184 E + 38 cannot be represented in BASIC's number format. Underflow is not a detectable error but less than 2.93873587 E-39 are indistinguishable from zero.

```
?1E40
?OVERFLOW ERROR
READY.
```

REDIM'D ARRAY--After a matrix was dimensioned, another dimension statement for the same matrix was encountered. For example, an array variable is defined by default when it is first used, and later a DIM statement is encountered.

```
A(5) = 6
DIM A(10,10)
?REDIM'D ARRAY ERROR
READY.
```

REDO FROM START--Is not actually a fatal error printed in the standard format but is a diagnostic printed when data in response to INPUT is alpha when a numeric quantity is required.

```
10 INPUT A
RUN
?ABC
?REDO FROM START
?
```

INPUT continues to function until acceptable data has been received. The complement to this diagnostic on files is BAD DATA ERROR which is fatal.When not enough data has been typed in response to INPUT, a double ? is printed until enough data is received.

```
10 INPUT A,B,C
RUN
?1
??2
??3
READY.
```

RETURN WITHOUT GOSUB--A RETURN statement was encountered without a previous GOSUB statement being executed.

```
CLR
RETURN
?RETURN WITHOUT GOSUB ERROR
```

STRING TOO LONG--Attempt by use of the concatenation operator to create a string more than 255 characters long.

```
A$ = 'A'
FOR I = 1 TO 10:A$ = A$ + A$:NEXT
?STRING TOO LONG ERROR
READY.
```

SYNTAX--BASIC cannot recognize the statement you have typed. Caused by such things as missing parenthesis, illegal characters, incorrect punctuation, mispelled keyword.

```
RUIN
?SYNTAX ERROR
READY.
```

TYPE MISMATCH--The left-handed side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function which expected a string argument was given a numeric one, or vice versa.

```
A$ = 5
?TYPE MISMATCH ERROR
READY.
```

UNDEF'D STATEMENT--An attempt was made to GOTO, GOSUB, or THEN to a statement which does not exist.

```
GOTO A
?UNDEF'D STATEMENT ERROR
READY.
```

UNDEF'D FUNCTION--Reference was made to a user defined function which had never been defined.

```
X = FNA(3)
?UNDEF'D FUNCTION ERROR
READY.
```

Operating System Messages and Meanings

BAD DATA--Numeric data was expected but alpha data was received when inputing from a special device.

DEVICE NOT PRESENT-- No device on the IEEE was present to handshake an attention sequence. Status will have a value of 2 which corresponds to a time out. May happen on OPEN, CLOSE, CMD, INPUT#, GET#, PRINT#

```
OPEN 5,4,3, 'FILE'
?DEVICE NOT PRESENT ERROR
READY.
```

FILE NOT FOUND--The named files specified in OPEN or LOAD was not found on the device specified. In the case of tape I/O, an end of tape mark was encountered. In disk I/O, the disk timed out when attempting to open the file, thus producing this message:

```
LOAD 'FILE', 15
?FILE NOT FOUND ERROR
READY.
```

FILE NOT OPEN--The operating system must have device number and command information provided by the OPEN statement. If an attempt is made to read or write a file without having done this previously, then this message appears:

```
CLR
INPUT#10,A
?FILE NOT OPEN ERROR
READY.
```

FILE OPEN--An attempt to redefine file parameter information by repeating an OPEN command on the same file twice.

```
OPEN 1,4,1
OPEN 1,4,1
?FILE OPEN ERROR
READY.
```

LOAD--Only occurs when loading a program from cassette tape. This means that there were more than 31 errors in the first tape block or that there were errors in exactly the same corresponding positions of both

blocks.

NOT INPUT FILE --Tape files, once opened for writing, cannot be read without first CLOSE rewinding tape and OPEN for INPUT. This message appears when an attempt is made to read on output file:

```
10 OPEN 1,1,1
20 INPUT #1,A
?NOT INPUT FILE ERROR
READY.
```

NOT OUTPUT FILE--Tape files cannot be read and updated in place. Device 0 is the keyboard and it cannot be written to:

```
10 OPEN 1,0
20 PRINT #1
?NOT OUTPUT FILE ERROR
READY.
```

VERIFY--The contents of memory and a specified file do not compare.

NOTES

Appendix A.

Detailed PET Memory Map

PET Memory Allocation By 4K Blocks

BLOCK #	TYPE	START ADDRESS	FUNCTION
*0	RAM	$0000	Working, text, variable storage.
1	RAM	$1000	Test variable storage (8K only)
2	---	$2000	Expansion RAM
3	---	$3000	Expansion RAM
4	---	$4000	Expansion RAM
5	---	$5000	Expansion RAM
6	---	$6000	Expansion RAM
7	---	$7000	Expansion RAM
8	RAM	$8000	Screen memory (1K)
9	---	$9000	Expansion ROM
10	---	$A000	Expansion ROM
11	---	$B000	Expansion ROM
12	ROM	$C000	BASIC (principally statement interpreter).
13	ROM	$D000	BASIC (principally math package).
*14	ROM I/O	$E000 $E800	Screen editor. All internal PET I/O.
15	ROM	$F000	OS diagnostics

*see expanded description

Block 0 By 256 Byte Pages

PAGE	TYPE	START ADDRESS	FUNCTION
**0	RAM	0000	BASIC OS working storage
**1	RAM	0100	Stack
**2	RAM	0200	O S working storage
**3	RAM	0300	Cassette buffers.
4-15	RAM	0400	BASIC text area

** see expanded description by page

Block 14 By 2K Segment

PAGE	TYPE	START ADDRESS	FUNCTION
0	ROM	$E000	Screen editor
1	I/O	$E800	PET I/O

I/O Device Base Addresses

PAGE	TYPE	START ADDRESS	FUNCTION
0	PIA	$E810	Keyboard
1	PIA	$E820	IEEE-488
2	VIA	$E840	USR PORT cassette

Location not specified are used but have no clear one function definition.

PET PAGE ZERO MEMORY MAP

FROM	TO	DESCRIPTION
000	--	$4C constant (6502 JMP instruction).
001	002	USR function address lo, hi.
Terminal I/O maintenance		
003	--	Active I/O channel #.
004	--	Nulls to print for CRLF (unused).
005	--	Column BASIC is printing next.
006	--	Terminal width (unused).
007	--	Limit for scanning source columns (unused).
008	--	Line number storage before buffer.
009	--	$2C constant (special comma for INPUT process).
010	089	BASIC INPUT buffer (80 bytes).
090	--	General counter for BASIC.
091	--	$00 used as delimeter.
092	--	General counter for BASIC.
Evaluation of variables		
093	--	Flag to remember dimensioned variables.
094	--	Flag for variable type; 0#numeric; 1 ÷ string.
095	--	Flag for integer tape.
096	--	Flag to crunch reserved words (protects '& remark).
097	--	Flag which allows subscripts in syntax.
098	--	Flags INPUT or READ.
099	--	Flag sign of TAN.
100	--	Flag to suppress OUTPUT (+ normal; − suppressed).
101	--	Index to next available descriptor.
102	103	Pointer to last string temporary lo; hi.
104	111	Table of double byte descriptors which point to vaiables.
112	113	Indirect index #1 lo; hi.
114	115	Indirect index #2 lo; hi.
116	121	Pseudo register for function operands.
Data storage maintenance		
122	123	Pointer to start of BASIC text area lo; hi byte.
124	125	Pointer to start of variables lo; hi byte.
126	127	Pointer to array table lo; hi byte.
128	129	Pointer to end of variables lo; hi byte.
130	131	Pointer to start of strings lo; hi byte.
132	133	Pointer to top string space lo; hi byte.
134	135	Highest RAM adr lo; hi byte.
136	137	Current line being executed. A zero in 136 means statement executed in a direct command.
138	139	Line # for continue command lo; hi.
140	141	Pointer to next STMNT to execute lo; hi.
142	143	Data line # for errors lo; hi.
144	145	Data statement pointer lo; hi.

Expression evaluation		
146	147	Source of INPUT lo; hi.
148	149	Current variable name.
150	151	Pointer to variable in memory lo; hi.
152	153	Pointer to variable referred to in current FOR-NEXT.
154	155	Pointer to current operator in table lo, hi.
156	--	Special mask for current operator.
157	158	Pointer to function definition lo; hi.
159	160	Pointer to a string description lo; hi.
161	--	Length of a string of above string.
162	--	Constant used by garbage collect routine.
163	--	$4C constant (6502 JMP inst).
164	165	Vector for function dispatch lo; hi.
166	171	Floating accumulator #3.
172	173	Block transfer pointer #1 lo; hi.
174	175	Block transfer pointer #2 lo; hi.
176	181	Floating accumulator #1. (USR function evaluated here).
182	--	Duplicate copy of sign of mantissa of FAC #1.
183	--	Counter for # of bits fo shift to normalize FAC # 1.
184	189	Floating accumulator #2.
190	--	Overflow byte for floating argument.
191	--	Duplicate copy of sign of mantissa.
192	193	Pointer to ASCII rep of FAC in conversion routine lo; hi.
RAM subroutines		
194	199	CHRGOT RAM code. Gets next character from BASIC text.
200	--	CHRGOT RAM code regets current characters.
201	202	Pointer to source text lo; hi.
203	223	Next random number in storage.
OS page zero storage		
224	225	Pointer to start of line of cursor loc lo; hi.
226	--	Column position of cursor.
227	228	General purpose start address indirect lo; hi.
229	233	General purpose and address direct lo; hi.
234	--	Flag for quote mode on/off.
238	--	Current file name length.
239	--	Current logical file number.
240	--	Current primary address.
241	242	Current secondary address.
243	244	Pointer to start of current tape buffer lo; hi.
245	--	Current screen line #.
246	--	Data temporary for I/O.
247	248	Pointer to start loc for O.S. lo; hi.
249	250	Pointer to current file name lo; hi.
251	254	Unused.
255	--	Overflow byte that BASIC uses when doing FAC to ASCII conversions.

Page 1

62 byte on bottom are used for error correction in tape reads. Also, buffer for ASCII when BASIC is expanding the FAC into a printable number. The rest of page 1 is used for storage of BASIC GOSUB and for NEXT context and hardware stack for the machine.

PET PAGE TWO MEMORY MAP

FROM	TO	DESCRIPTION
512	514	24-hour clock in 1/60 sec.
517	518	Correction factor for clock LSB; MSB.
519	520	Interrupt driver flag for cassette #1. switches; #2 switches.
523	--	Flag≠ means verify not load into memmory.
524	--	I/O status byte.
525	--	Index into keystroke buffer.
526	--	Flag to indicate reverse-field on.
527	536	Interrupt driven key stroke buffer.
537	538	IRQ RAM VECTOR lo; hi.
539	540	BRK instruction RAM VECTOR lo; hi.
549	--	Count down to flip cursor.
551	--	Flag for cursor on/off.
553	577	Table of LSB of start addresses of video dislay lines (25).
578	587	Table of logic addresses.
588	597	Table of primary addresses.
598	609	Table of secondary addresses.
610	--	Index into LA, FA, SA tables.
611	--	Default input device #.
612	--	Default output device #.
613	--	Computation of parity on cassette write.
621	--	Count of redundant tape blocks.
624	--	Count down synchronization or cassette write.
625	626	Index next character in/out tape buffer #1; #2.
627	--	Countdown synchronization on cassette read.
628	--	Flag to indicate bit/byte tape error.
629	--	Flag to indicate tape routine reading shorts.
630	631	Index to addresses to correct on tape read pass 1; pass 2.
632	--	Flag for cassette read--tells current function--countdown, read, etc.
633	--	Count of seconds of shorts to write before data.
634	825	Buffer for cassette #1 (192 bytes).
826	1017	Buffer for cassette #2 (192 bytes).
1018	1023	Unused.

VARIABLE ALLOCATION

Space is allocated for variables only as they are encountered. It is not possible to allocate an array on the basis of 2 single elements, hence the reason to execute DIM statement before array references. Seven bytes are allocated for each simple variable whether it is a string, number, or user defined function.

The first two bytes give the name of the variable:

	byte 1	byte2
INTEGER	first chr + 128	Second chr + 128 or 128
FLOATING	first chr	second chr or 0
STRING	first chr	second chr + 128 or 128

The last five bytes give the value of a variable, or a descriptor to the rest of the data:

INTEGER

actual value				
256 * HI	LO	0	0	0

FLOATING

actual value in binary floating point				

STRING

	pointer			
chr count	LO	HI	0	0

The simple string variable points to a location in high memory, where the actual characters are stored.

Examples of declaration and storage

```
15%=90
     201  181  0  90  0  0  0
C$="HELLO"
      67  128  5  .  .  0    0
```

H	E	L	L	O

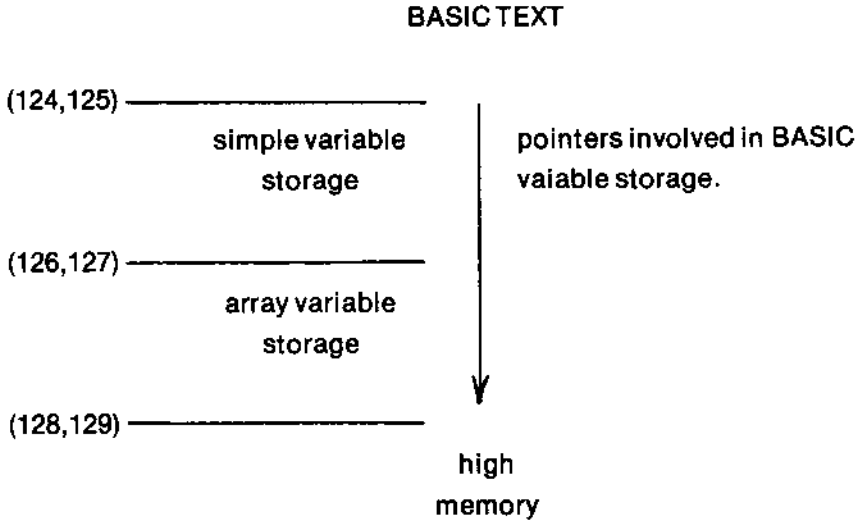
Locations 124 and 125 contain the first address of memory where a simple variable name will be found.By incrementig the address by 7 each time the ext simple variable name in the table is encountered.The end of the variables is defined-by the address in 126 and 127.

Locations 126 and 127 also define the start of array storage. The first two bytes of array descriptors arethe same as simple variables but the next five bytes are special as follows:

	byte 3	byte 4	byte 5	byte 6	byte 7
VECTOR ARRAYS	7 + (size + 1)* (dim)*A	0	1	0	size + 1

where A = 2 for integer, = 3 for string, or = 5 for floating.

By incrementing the search address by the current byte #3 of the descriptor each time, the next array variable is reached. Locations 128 and 129 contain the ending address of this table.



Because the variables are divided in storage between arrays and simple variables insertion of an additional simple variable is a bit more complicated once an array has been defined. First, the entire array storage area must be block moved upward by seven bytes and the pointers adjusted upward +7. Finally, the simple variable can be inserted at the end of simple variable storage.

> If large arrays are defined and initialized first before simple variables are assigned, much execution time can be lost moving the arrays each time a simple variable is defined. The best strategy to followin this case is to assign a value to all known simple variables before assigning arrays. This will optimize execution speed.

Functions of NEW and CLR on data pointer:

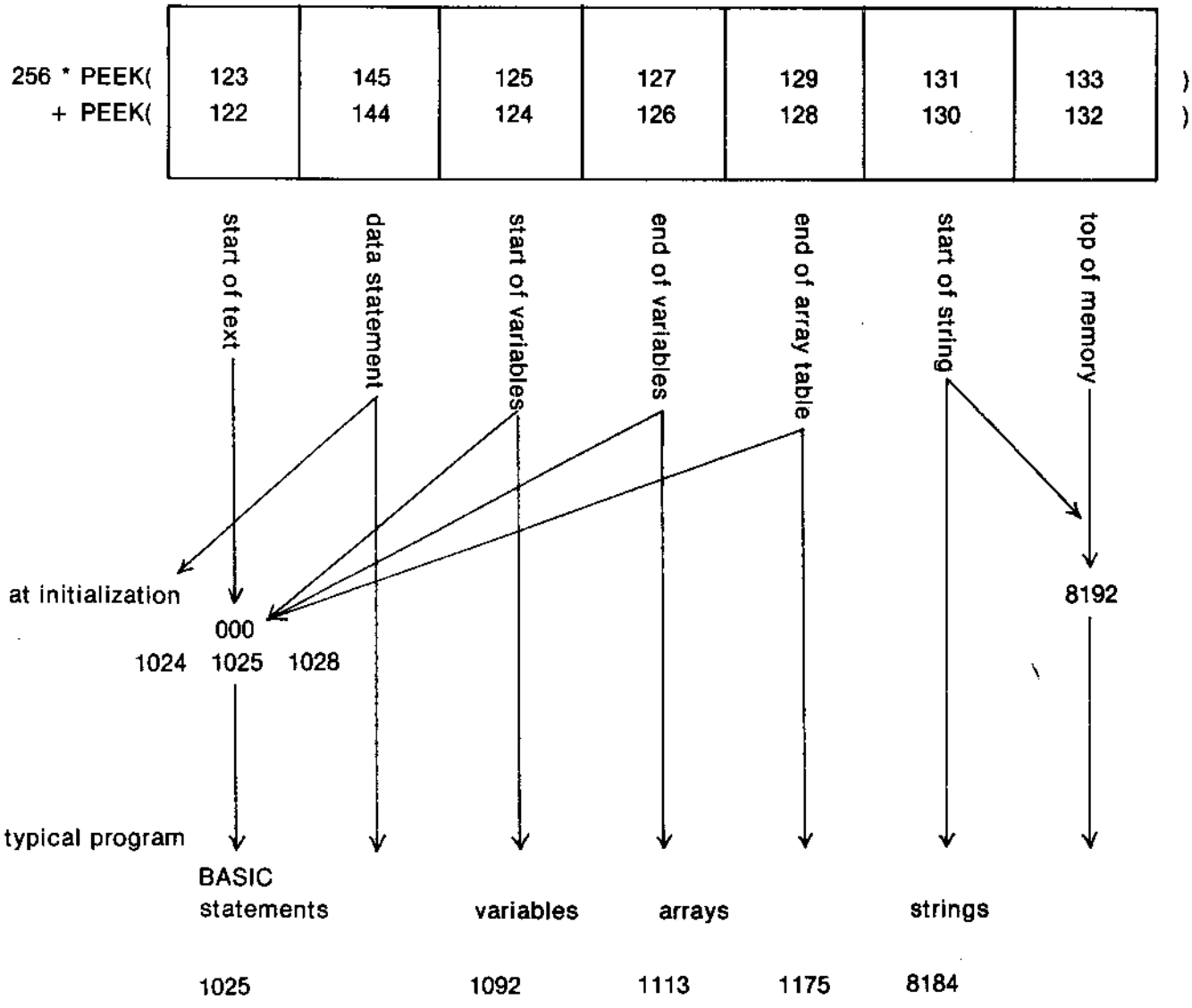
C L R

String pointer equated to top of memory data pointer to start of text − 1 end of array table to start of variables end of simple variables to start of variables.

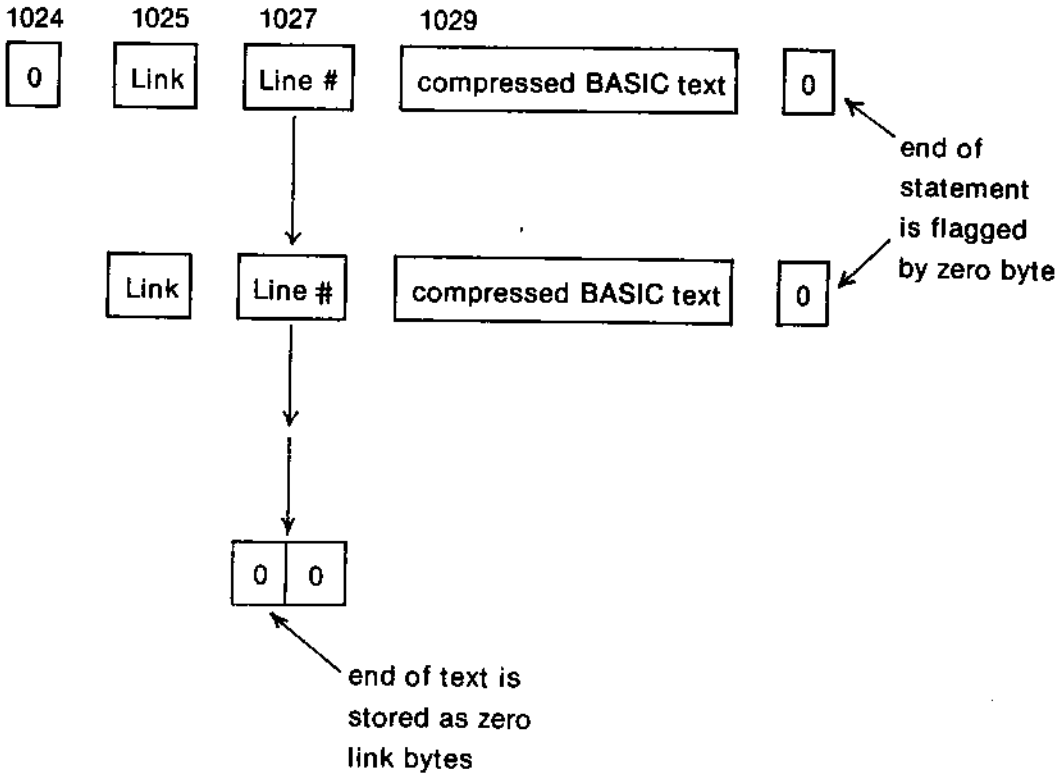
N E W

String pointer equated to top of memory data pointer to start of text − 1 end of array table to start of text +3 end of simple variables to start of text + start of variables to start of text +3.

PRINCIPAL POINTERS INTO PET RAM



HOW BASIC STATEMENTS ARE STORED



BASIC STATEMENTS

DEF FN
DIM
END
FOR-TO-STEP-NEXT
GET
GOSUB-RETURN
GOTO
IF-THEN
INPUT
LET
ON-(GOSUB-GOTO)
POKE-PEEK
PRINT
READ-DATA-RESTORE
REM
STOP-CONT
WAIT

In the following description of statements, an argument of V or W denotes a numeric variable. X denotes a numeric expression, X$ denotes a string expression and an I or J denotes an expression that is truncated to an integer begore the statement is executed. Truncation means that any fractional part of the number is lost, e.g. 3.9 becomes 3, 4.01 becomes 4.

DEF

100 DEF FNA (V) = V/B + C

The user can define functions like the built-in functions (SQR, SGN, ABS, etc) through the use of the DEF statement. The name of the function is 'FN' followed by any legal variable name, for example: FNX, FNJ7, FNKO, FNR2. User-furnished functions are restructed to one line.
A function may be defined to be any expression, but may only have one argument. In the example, B & C are variables that are used in the program. Executing the DEF statement defines the function. User-defined functions can be redefined by executing another DEF statement for the same function.
User-defined string functions are not allowed. 'V' is called the dummy variable.

	110 Z = FNA(3)	Execution of this statement following the above would cause Z to be set to 3/B + C, but the value of V would be unchanged.
	200 DEF FNA(V) = FNB(V)	A function definition may be recursive. A DEF statement may be written in terms of other functions, however.
DIM	113 DIM A(3),B(10)	Allocates space for matrices. All matrix examples are set to zero by the DIM statement.
	114 DIM R3(5,5), D$(2,2,2)	Matrices can have more than one dimension. Up to 255 elements
	115 DIM Q1(N),Z(2*1)	Matrices can be dimensioned dynamically during program execution. If a matrix is not explicitly dimensioned with a DIM statement, it is assumed to have as many subscripts as implied in its first use and whose subscripts may range from 0 to 10 (eleven elements).
	117 A(8) = 4	If this statement was encountered before a DIM statement for A was found in the program, it would be as if a DIM A(10) had been executed previous to the execution of line 117. All subscripts start at zero (0), which means that DIM × (100) really allocates 101 matrix elements.
END	999 END	Terminates program execution without printing a BREAK message. (See STOP) CONT after an END statement causes execution to resume at the statement after the END statement. END can be used anywhere in the program, and is optional.
FOR	300 FOR V = 1 TO 9.3 STEP .6	V is set equal to the value of the expression following the equal sign, in this case 1. This value is called the initial value. Then the statements between FOR and NEXT are executed. The final value is the value of the expression following the TO. The step is the value for the expression following STEP. When the NEXT statement is encountered, the step is added to the variable.
	310 FOR V = 1 TO 9.3	If no STEP was specified, it is assumed to be one. If the step is positive and the new value of the variable is < = to the final value (9.3 in this example), or the step value

		is negative and the new value of the variable is =>the final value, then the first statement following the FOR statement is executed. Otherwise, the statement following the NEXT statement is executed. All FOR loops execute the statements between the FOR and the NEXT at least once, even in the case like FOR V = 1 TO 0.
	315 FOR V = 10*N TO 3.4/Q STEP SQR(R)	Note that expressions (formulas) may be used for the initial, final and step values in the FOR loop. The variables of the expressions are computed only once, before the body of the FOR...NEXT loop to terminate. The statement between the FOR and its corresponding NEXT in both example above (310) would be executed 9 times.
	340 NEXT V	Marks the end of a FOR loop.
	345 NEXT	If no variable is given, matches the most recent FOR loop.
	350 NEXT V,W	A single NEXT may be used to match multiple FOR statements. Equivalent to NEXT V: NEXT W. Specification the former way saves 1 byte of BASIC text storage.
GET	GET A GET A$	Works like INPUT or INPUT# on a single character basis. Unlike INPUT though, this function scans the keyboard and does not wait for carriage return to be pressed. If no key has been pressed, A$ = " "(null string) and A = 0 after executing this statement.
	10 GET A$: 1FA$ = " "THEN 10	This example stays in a loop until a key has been pressed.
GOSUB	10 GOSUB 910	Branches to the specified statement (910) until a RETURN is encountered; when a branch is then made to the statement after the GOSUB. GOSUB nesting is limited to 23 levels. Subroutines line numbers are searched for from the beginning of text. To increase execution speed, define subroutines first with low line numbers. Fewer digits in line numbers will also save storage space.

	50 RETURN	Causes a subroutine to return to the statement after the most recently executed GOSUB.
GOTO	50 GOTO 100	Branches to the statement specified. Keeping line numbers low will save space on GOSUB statements.
IF...GOTO	32 IF x< = Y + 23x4 GOTO 92	Equivalent to IF...THEN, except that IF... GOTO must be followed by a line number, while IF...THEN can be followed by either a line number or another statement.
IF...THEN	15 IF x<0 THEN 5	Branches to specified statement if the relation is True.
	25 IF X = 5 THEN 50:Z = A	WARNING. The "Z = A" will never be executed because if the relation is true, BASIC will branch to line 50. If the relation is a false, BASIC will proceed to the line after line 25.

26 IF X<0 THEN PRINT "ERROR X NEGATIVE": GOTO 350

In this example, if X is less than 0, the PRINT statement will be executed and then the GOTO statement will branch to line 350. If the X was 0 or positive, BASIC will proceed to execute the lines after line 26. Binary floating point representations of decimal fractions may not alwys be exact. sometimes a comparison will fail because of this. In this case, compare the number to a ± range.

INPUT		Request information character by character until carriage return from the keyboard, turning the characters into numbers or strings of a maximum length of 79 characters.
	3 INPUT V,W,W2	Requests data from the terminal (to be typed in). Each value must be separated from the preceeding value by a comma (,). The last value typed should be followed by a carriage return. A "?" is typed as a prompt character. However, only constants may be typed in as a response to an INPUT statement, such as 4.5E-3 or "CAT". If more data was requested in an INPUT statement than was typed in, a "??" is printed (if INPUT is from terminal) and the rest of the data should be typed in. If more

		data was typed in than requested, the extra data will be ignored and a warning "EXTRA IGNORED" will be printed when this happens. String must be input in the same format as they are specified in DATA statements.
	5 INPUT "VALUE";V	Optionally types a prompt string ("VALUE") before requesting data from the terminal. Typing CONT after an INPUT command has been interrupted will cause execution to resume at the INPUT statement. An INPUT command is interrupted if a carriage return is the only character entered.
LET	300 LET W = X 310 V = 5.1	Assigns a value to a variable. "LET" is optional. The type of variable (numeric or string) must be the same as the evaluated expression.
ON...GOTO	100 ON I GOTO 10,20,30,40	Branches to the line indicated by the I'th number after the GOTO. That is : If I = 1, THEN GOTO LINE 10 If I = 2, THEN GOTO LINE 20 If I = 3, THEN GOTO LINE 30 If I = 4, THEN GOTO LINE 40. If I = 0 or I attempts to select a nonexistent line (> =) in this case, the statement after the ON statement is executed. However, if I is <255 or >0, an "ILLEGAL QUANTITY" error message will result. As many line numbers as will fit on a 79-byte line can follow an ON...GOTO.
	105 ON SGN (X) + 2 GOTO 40,50,60	This statement will branch to line 40 if the expression X is less than zero, to line 50 if it equals zero, and to line 60 if it is equal to one.
ON...GOSUB	110 ON 1 GOSUB 50,60	Identical to "ON...GOTO", except that a subroutine called (GOSUB), is executed instead of a GOTO. RETURN from the GOSUB branches to the statement after the ON...GOSUB.
POKE	357 POKE I,J	The POKE statement stores the byte specified by its second argument (J) into the location given by its first argument (I).

		The byte to be stored must be =>0 and <=255, or an "ILLEGAL QUANTITY" error will occur. The address (I) must be =>0 and <=65535, or an "ILLEGAL QUANTITY" error will result. POKE works only on RAM and I/O POKEing. Certain locations will disturb normal PET operation unless reset. It is not possible to POKE the PEEK of a location into a location inPET ROM.
PEEK	10A=PEEK(I)	PEEK is a function of an address and returns a byte value contained in that location. BASIC cannot be PEEKed and PEEK of locations $C000 to $E1D9 yields a value of zero.
PRINT		Sends the data to PET TV display. BASIC software calls a subroutine in the system software and loads the character in the accumulator.
	360 PRINT X,Y,Z 370 PRINT 380 PRINT X,Y 390 PRINT "VALUE" IS";A 400 PRINT A2,B,	Prints the value of expressions on the terminal. If the list of values to be printed out does not end with a comma (,) or a semicolon (;), then a carriage return/line feed is executed after all the values have been printed. Strings enclosed in quotes (") may also be printed. If a semicolon separates two expressions in the list, their values are printed nest to each other. If a comma appears after an expression in the list, then spaces are printed until the carriage is at the beginning of the next N column field (until the carriage is at column N,2N,3N,4N...). If there is no list of expressions to be printed, then a carriage return is executed.
	410 PRINT MID$(A$,2);	String expressions may be printed. A semicolon is not neeeded between string expressions such as PRINT AB "HELLO" that are to be concatenated.
READ	490 READ V,W	Reads data into specified variable from a DATA statement. The first piece of data read will be the first piece of data listed in the first data statement of the program. The second piece of data read will be the second piece listed in the first DATA statement, and so on. When all of the data

		have been read from the first DATA statement, the next piece of data to be read will be the first piece listed in the second DATA statement of the program. Attempting to read more data then there is in all the DATA statements in a program will cause an "OUT OF DATA" error. The line number given in the "SYNTAX ERROR" will refer to the line number where the error actually is located.
DATA	10DATA1,3, – 1E3,.04	Specifies data, read from left to right. Information appears in data statements in the same order as it will be read in the program.
	20 DATA "CBM,INC" 30 DATA PET, "2001"	Strings may be read from DATA statements. If you want the string to contain a colon (:) or commas (,), or leading blanks, you must enclose the string in double quotes. It is impossible to have a double quote within string data or a string literal. (" "ANYTHING" ") is illegal.
RESTORE	510 RESTORE	Allows the rereading of DATA statements. After a RESTORE, the next piece of data read will be the first piece listed in the first DATA statement, and so on as in a normal READ operation.
REM	500 REM NOW SET V = 0	Allows the programmer to put comments in his program. REM statements are not executed, but can be branched to. A REM statement is terminated by end of line, but not by a ":".
	505REM SET V = 0: V = 0	In this case, the V = 0 will never be executed by BASIC.
	506 V = 0: REM SET V = 0	In this case V = 0 will be executed.
STOP	9000 STOP	Causes a program to stop execution and to enter command mode. Prints BREAK IN LINE 9000 (as per this example). CONT after a STOP branches to the statement following the STOP.
CONT		A command that can be executed only in direct mode. Resumes program execution after STOP, END, or use of STOP key. A program cannot be resumed after error condition, editing, CLR, or NEW.
WAIT	WAIT I,J,K	This statement reads the status of memory

location I, exclusive OR's K with status, then AND's the result with J untill a non-zero result is obtained. Execution of the program continues at the statement following the WAIT.
If the WAIT statement only has two arguments, K is assumed to be zero. If you are waiting for a bit to become zero, there should be a one in the corresponding position of K. 0<= I<=65536 J,K must be <=0 and >=255.
The STOP key cannot interrupt a WAIT.

Appendix C

BASIC COMMANDS

CLR
LIST
LOAD
NEW
RUN
SAVE
VERIFY

A command is usually given after BASIC has typed READY. This is called the "Command Level". Commands may be used as program statements. Certain commands, such as LIST and NEW will terminate program execution when they finish.

CLR		Deletes all stored references to variables, arrays, functions, GOSUB and FOR-NEXT context.
LIST	LIST X	Lists line "X" if there is one.
	LIST or LIST-	Lists the entire program.
	LIST X-	Lists all lines in a program with a line number equal to, or greater than, "X".
	LIST -X	Lists all of the lines in a program with a line number less than, or equal to, "X".
	LIST Y-X	Lists all of the lines within a program with line numbers equal to, or greater than, "Y", and less than or equal to "X".
		If LIST is used as a program statement, the program will terminate after it is executed.
LOAD	LOAD	Load first program found on cassette #1 into memory.
	LOAD "HURKLE"	Search for named file on cassette #1 and then load it into memory.
	LOAD "HURKLE", 2	Same as previous, except from device #2.
	10 LOAD "HURKLE"	When LOAD is specified as a program statement, execution of the current program in memory stops at this point. A normal load of program proceeds. The new program begins execution from its lowest line number. Variables and their values are passed from the load to the new program. Strings and function definitions cannot be relied upon because BASIC maintains pointers into the old text

		where they used to be. Strings can be forced to exist in permanent string variable storage by performing an operation on them prior to LOAD, e.g. A$ = A$ + " ". WARNING: On an overlay LOAD, the overlaying program must have a text storage requirement less than or equal to the previous program. If this is not true, then the variables will be overwritten because they are stored immediately after text in memory.
NEW		Deletes current program and all variables.
RUN	RUN	Starts execution of the program currently in memory at the lowest numbered statementment. RUN deletes all variables (like CLR) and restores DATA. If you have stopped your program and wish to continue execution at some point in the program, use a direct GOTO statement to start execution of your program at the desired line.
	RUN 200	Optionally starts RUN at the specified line number.
SAVE	SAVE	Save BASIC text on cassette #1.
	SAVE "HURKLE"	Save and name the file on cassette #1.
	SAVE "HURKLE", 2	Save on 2nd cassette unit.
	SAVE "HURKLE", 2,1	Save and write end of tape block.
VERIFY	VERIFY "HURKLE"	Same parameters as LOAD. Compares contents of memory with file and reports success/failure of compare.

EXPRESSIONS AND OPERATORS

RELATIONAL OPERATORS

=	equal
<	less than
>	greater than
<=	L.E.
>=	G.E.
<>	not equal

BOOLEAN OPERATORS

AND
OR
NOT

ARITHMETIC OPERATORS

+	add
–	subtract
*	multiply
/	divide
↑	exponentiation
–	(negation)

STRING OPERATOR

+	(concatenation)

ARITHMETIC OPERATORS

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	A = 100 LET Z = 2.5	Assigns a value to a variable, the LET is optional.
-	B = - A	Negation. Note that 0 - *A is subtraction, while* - A is negation.
↑	130 PRINT X↑3	Exponentation (equal to X*X*X in the sample statement). 0↑0 = 1. 0 to any other power = 0.A↑B, with A negative and B not an integer gives an FC error.
*	140 X = R*(B*D)	Multiplication.
/	150 PRINT x/1.3	Division.
+	160.Z = R + T + Q	Addition.
-	170 J = 100 - I	Subtraction.

RELATIONAL OPERATORS

Relational operators can be used as part of any expression.
Relational operator expressions will always have a value of True (- 1) or a value of False (0).
Therefore, (5 = 4) = 0, (5 = 5) = - 1 , etc.
The THEN clause of an IF statement is executed whenever the formula after the IF is not equal to 0. That is to say, IF X THEN...is equivalent to IF X < > 0 THEN....

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	10 IF A = 15 THEN 40	Expression Equals Expression.
< >	70 IF A < > 0 THEN 5	Expression Does Not Equal Expression.
>	30 IF B > 100 THEN 8	Expression Greater Than Expression.
<	160 IF B < 2 THEN 10	Expression Less Than Expression.
< =, =<	180 IF 100 < = B + C THEN 10	Expression Less Than Or Equal To Expression.
> =, =>	190 IF Q > = R THEN 50	Expression Greater Than Or Equal To Expression.

BOOLEAN OPERATORS

AND	2 IF A < 5 AND B < 2 THEN 7	If expression 1 (A < 5) AND expression 2 (B < 2) are both true, then branch to line 7.
OR	IF A < 1 OR B < 2 THEN 2	If either expression 1 (A < 1) OR expression 2 (B < 2) is true, then branch to line 2.
NOT	IF NOT Q3 THEN 4	If expression "NOT Q3" is true (because Q3 is false), then branch to line 4. NOT - 1 = 0 (NOT true = false).

AND, OR and NOT can be used for bit manipulation, and for performing boolean operations.
These three operators convert their arguments to sixteen bit, signed two's, complement integers in the

range −32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not in this range, an ?ILLEGAL QUANTITY ERROR results. The operations are performed in bitwise fashion, this means that each bit of the result is obtained by examining the bit in the same position for each argument.
The following truth table shows the logical relationship between bits:

OPERATOR	ARG. 1	ARG. 2	RESULT
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
NOT	1	-	0
	0	-	1

EXAMPLES OF BOOLEAN EXPRESSIONS

63 AND 16 = 16	Since 63 equals binary 111111 and 16 equals binary 10000, the result of the AND is binary 10000 or 16.
15 AND 14 = 14	15 equals binary 1111 and 14 equals binary 1110, so 15 and 14 equals binary 1110 or 14.
−1 AND 8 = 8	−1 equals binary 1111111111111111 and 8 equals binary 1000, so the result is binary 1000 or 8 decimal.
4 AND 2 = 0	4 equals binary 100 and 2 equals binary 10, so the result is binary 0 because none of the bits in either argument match to give a 1 bit in the result.
10 OR 10 = 10	Binary 1010 OR'd with binary 1010, or 10 decimal.
−1 OR −2 = −1	Binary 1111111111111111 (−1) OR'd with binary 1111111111111110 (−2) equals binary 1111111111111111, or −1.
NOT 0 = −1	The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or −1. Also NOT −1 = 0.
NOT X	NOT X is equal to −(X + 1). This is because to form the sixteen bit two's complement of the binary, you take the bit (one's) complement and add one.
NOT 1 = −2	The sixteen bit complement of 1 is 1111111111111110, which is equal to −(1 + 1) or −2.

RULES FOR EVALUATING EXPRESSIONS

Rules for Evaluating Expressions:

1. Operations of higher precedence are performed before operations of lower precedence. This means the multiplications and divisions are performed before additions and subtracions. As an example, 2 + 10/5 equals 4, not 2.4. When operations of equal precedence are found in a formula, the left-hand one is executed first: 6 − 3 + 5 = 8, not −2.

2. The order in which operations are performed can always be specified explicitly through the use of parentheses. For instance, to add 5 to 3 and then divide that by 4, we would use (5 + 3)/4, which eqals 2. If, instead, we had used 5 + 3/4, we would get 5.75 as a result (5 plus 3/4).

The precedence of operators used in evaluating expressions is as follows, in order beginning with the highest precedence: (Note: Operators listed on the same line have the same precedence).

1) FORMULAS ENCLOSED IN PARENTHESIS ARE ALWAYS EVALUATED FIRST
2) ↑ EXPONENTATION
3) NEGATION −X WHERE X MAY BE A FORMULA
4) * / MULTIPLICATION AND DIVISION
5) + − ADDITION AND SUBTRACTION
6) RELATIONAL OPERATORS: = EQUAL
<> NOT EQUAL
(equal precedence < LESS THAN
for all six). > GREATER THAN
<= LESS THAN OR EQUAL
>= GREATER THAN OR EQUAL
7) NOT LOGICAL AND BITWISE "NOT" LIKE NEGATION, NOT TAKES ONLY THE FORMULA TO ITS RIGHT AS AN ARGUMENT
8) AND LOGICAL AND BITWISE "AND"
9) OR LOGICAL AND BITWISE "OR"

Appendix E

SPACE HINTS

In order to make your program smaller and save space, the following hints may be useful.

1) Use multiple statements per line. There is a small amount of overhead. (5 bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 63999), it takes the same number of bytes. Putting as many statements as possible in a line will cut down on the number of bytes used by your program.

2) Delete all unnecessary spaces from your program. For instance:

10 PRINT X, Y, Z

uses three more bytes than

10 PRINTX,Y,Z

Note: All spaces between the line number and the first non-blank character are ignored.

3) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the text. For instance, the statement 130 REM THIS IS A COMMENT uses up 24 bytes of memory.

In the statement 140 X = X + Y:REM UPDATE SUM, the REM uses 14 bytes of memory including the colon before the REM.

4) Use variables instead of constants. Suppose you use the constant 1.02369 ten times in your program. If you insert a statement

10Q = 1.02369

in the program, and use Q instead of 1.02369 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

5) A program need not end with an END; so, an END statement at the end of a program may be deleted.

6) Re-use the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use it again. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7) Use GOSUB's to execute sections of program statemnts that perform identical actions.

8) Use the zero elements of matrices; for instance, A(O), B(O,X)

SPEED HINTS

The hints below should improve the execution time of your BASIC program. Note that some of these hints are the same as those used to decrease the space used by your programs. This means that in many cases you can increase the efficiency of both the speed and size of your programs at the same time.

1) Delete all unnecessary spaces and REM's from the program. This may cause a small decrease in execution time because BASIC would otherwise have to ignore or skip over spaces and REM statements.

2) THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10. Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly.

3) Order your definitions of variables carefully. Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as 5 A = O:B = A:C = A, will place A first, B second, and C third in the symbol table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the symbol table to find A, two entries to find B and three entries to find C, etc.

4) Use NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT is the same variable in the most recent FOR statement.

Appendix F

MAIN LOGIC ASSEMBLY PARTS CROSS REFERENCE

REF. DES.	DESCRIPTION	PART NO.
C1,C2,C14	.O1MF 100V Ceramic	900010-38
C3	82 pF 500V Ceramic	900010-40
C4	10pF 500V Ceramic	900010-35
C5,C11,C12,C13	1.0MF 25V Tantalum	900402-13
C6	.1MF 50V Ceramic	900010-20
C7-C10	47MF 16V Electrolytic	900100-33
C15-C49	.1MF 10V Ceramic	900010-39
CR1,CR2,CR3	IN5402 3A/200V	900753-01
DS1	LED Indicator	900701-01
J5	20 Pin Header (Molex)	903307-02
J7	7 Pin Header (Molex)	903307-02
J8	5 Pin Header (Molex)	903302-02
Q1,Q4	TIP 29	902653-01
Q2,Q5	2N 4401	902652-01
Q3,Q6	2N 3904	902658-01
R1,R4	1.5K 1/4W 5%	901550-69
R2,R6,R12,R13,R14	10K 1/4W 5%	901550-20
R18-R25	10K 1/4W 5%	901550-20
R3,R5,R26-R46	1K 1/4W 5%	901550-01
R7,R8,R9	470 1/4W 5%	901550-58
R10	2.4K 1/4W 5%	901550-85
R11	5.1K 1/4W 5%	901550-03
R15,R16	IM 1/4W 5%	901550-84
R17	3.3K 1/4W 5%	901550-02
UA1,UB5,UC9	74LS93 Counter	901521-07
UA2	6540-010 MOS Char. Gen.	901439-08
UA2	2316B-08 Char Gen.	901447-08
UA5	6522 VIA	901437-01
UA7,UA8,UA9	MC 3446 Interface Bus	901524-01
UB1,UE8,UE9	74LS20 Nand Gate	901521-04
UB2	74LS165 Shift Reg.	901521-12
UB3,UB4,UG5,UG6, UH8,UH9	74LS244 Buffer	901521-13
UB6,UC5,UC7,UC8,UD5	74LS107 Flip-Flop	901521-08
UB8,UG8	6520 PIA	901436-01
UC1	74LS74 Flip-Flop	901521-06
UC2,UD8,UG3	74LS00 Nand Gate	901521-01
UC3,UC4,U11-U18 UJ1,UJ8	6550 RAM	901438-01

UC3,UC4,U11-U18 UJ1,UJ8	2114, RAM	901453-01
UC6,UE2	74LS08	901521-03
UD2,UD3,UD4	74LS157 Data Sel	901521-11
UD6,UD7	74177 Counter	901522-03
UD9,UE5	74LS04 Hex Inv	901521-02
UDE9	LM555 Timer	901523-01
UE3,UE4	7417 Hex Buffer	901522-01
UE6	74100 Latch	901522-02
UF3	6502 MPU	901435-01
UG2	74LS154 Decoder	901521-10
UG4	74LS21 And Gate	901521-05
UG9	74LS145	901521-09
UH1	6540-011 ROM	901439-01
UH1	2316B-01 ROM	901447-01
UH2	6540-013 ROM	901439-02
UH2	2316B-03 ROM	901447-03
UH3	6540-015 ROM	901439-03
UH3	2316-B-05 ROM	901447-05
UH4	6540-016 ROM	901439-04
UH4	2316B-06 ROM	901447-06
UH5	6540-012 ROM	901439-05
UH5	2316B-02 ROM	901447-02
UH6	6540-014 ROM	901439-06
UH6	2316B-04 ROM	901446-04
UH7	6540-018 ROM	901439-07
UH7	2316B-07 ROM	901446-07
UA2,UH1-UH7	Socket28 PIN	904153-05
UA2,UH1-UH7	Socket 24PIN	904153-04
UA5,UB8,UF3,UG8	Socket 40PIN	904153-06
UC3,UC4,UJ1-UJ8, U11-U18	Socket 22PIN	904153-03

Appendix G

SUGGESTED READING (USA produced)

Entering BASIC. J.Sack and J. Meadows. Science Research Associates,1973

BASIC:A Computer Programming Language. C. Pegels, Holden-Day,Inc. 1973

BASIC Programming. J. Kemeny and T. Kurtz, Peoples Computer Co., 1010 Doyle(P.O.Box 3100), Menlo Park, Ca 94025, 1967

BASIC. Albrecht, Finkle and Brown. Peoples Computer Co., 1010 Doyle(P.O.Box 3100), Menlo Park, Ca 94025, 1973

A Guided Tour of Computer Programming in BASIC. T. Dwyer, Houghton Mifflin Co., 1973

Programming Time Shared Computer in BASIC. Eugene H. Barnett. Wiley-Interscience L/C 72-175789 ($12.00)

Programming Language #2. Digital Equipment Corp., Maynard, MA 01754

101 BASIC Computer Games. Software Distribution Center. Digital Equipment Corp., Maynard, MA01754 ($7.50)

What to Do After You Hit Return. Peoples Computer Co., 1010 Doyle(P.O.Box 310), Menlo Park, Ca 94025 ($6.95)

Basic BASIC. James S. Coan, Hyden Book Co., Rochelle Park, NJ

WORKBOOKS 1-5. T. I. S., P.O.Box 921, Los Almos, NM 87544

Commodore Business Machines, Inc.
901 California Avenue
Palo Alto, California 94304, USA

Commodore/MOS
Valley Forge Corporate Center
950 Rittenhouse Road
Norristown, Pennsylvania 19401, USA

Commodore Business Machines Limited
3370 Pharmacy Avenue
Agincourt, Ontario, Canada M1W2K4

Commodore Business Machines (UK) Limited
360 Euston Road
London NW1 3BL, England

Commodore Buromaschinen GmbH
Frankfurter Strasse 171-175
6078 Neu Isenburg
West Germany

Commodore Japan Limited
Taisei-Denshi Building
8-14 Ikue 1-ChomeAsahi-Ku, Osaka 535, Japan

Commodore Electronics (Hong Kong) Ltd.
Watsons Estates
Block C, 11th floor
Hong Kong, Hong Kong